# FORMULAIRE D'INFORMATIQUE

**Classe de PCSI - Année 2025-2026**


*Python et Spyder : histoire et fonctionnement de l'interface (p.2)*
*Variables et opérations informatiques (p.4)*
*Entiers et flottants (p.5)*
*Tableaux et listes (p.6)*
*Tuples (p.7)*
*Chaînes de caractères (p.8)*
*Dictionnaires (p.9)*
*Instructions conditionnelles (p.10)*
*Instructions itératives (p.12)*
*Fonctions Python (p.13)*
*Algorithmes classiques (p.14)*
*Autres algorithmes assez classiques (p.15)*
*Bibliothèques et modules (p.16)*
*Bibliothèque* `matplotlib.pyplot` *(p.17)*
*Algorithmes dichotomiques (p.18)*
*Algorithmes récursifs (p.20)*
*Algorithmes de tri (p.23)*
*Algorithmes gloutons (p.28)*
*Manipulation de fichiers texte (p.29)*
*Manipulation de fichiers image (p.30)*
*Représentation des nombres (p.33)*
*Méthodes de programmation (p.41)*
*Terminaison et correction d'un algorithme (p.42)*
*Complexité (p.45)*
*Graphes : Généralités (p.49)*
*Graphes : Parcours de graphes (p.53)*
*Graphes : Plus court chemin (p.56)*

# PYTHON ET SPYDER

## ALGORITHME

Un **algorithme** est une suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution de problème.

**Faire de l'algorithmique**, c'est donc décomposer un problème complexe en des sous-problèmes de plus en plus simples jusqu'à arriver aux actions élémentaires que connaît l'ordinateur.

## LE LANGAGE DE PROGRAMMATION PYTHON

Une fois conçu, on traduit l'algorithme dans un **langage de programmation** (Java, C, C++, PHP, Python, Maple, Scilab, Pascal, etc) qui convertit les instructions en langage machine et permet ainsi à l'ordinateur d'exécuter l'algorithme.

Attention : un langage de programmation n'est pas un logiciel !

**Python** est le langage de programmation au programme des CPGE scientifiques. Ce langage a été développé par le mathématicien hollandais **Guido Von Russom** à la fin des années 80 - début des années 90. Celui-ci a nommé le langage en référence à la troupe d'humoristes britanniques des **Monthy Python**.

Python est un langage de **haut niveau**, c'est-à-dire un langage de programmation orienté vers les problèmes à résoudre, permettant d'écrire facilement des programmes à l'aide de mots usuels (en anglais) et de symboles mathématiques. A contrario, un langage de **bas niveau** se rapproche du langage machine (dit binaire) et permet de programmer à un niveau très avancé, ce qui induit des temps de calculs réduits pour un problème donné par rapport à un langage de haut niveau. La contrepartie dans l'utilisation d'un langage de bas niveau est la longueur du code qui est en général bien plus importante.

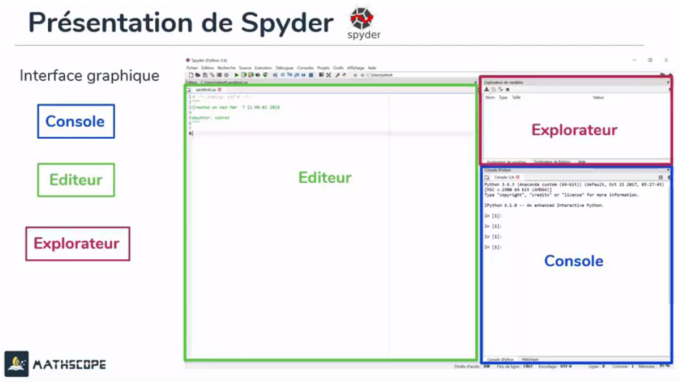
## L'INTERFACE SPYDER

Pour travailler en Python, il suffit d'écrire de simples fichiers textes (extension « *.py ») et de les interpréter. Cependant, pour faciliter l'interface Python-utilisateur on utilise souvent un environnement de développement pour faciliter la programmation comme les logiciels Pyzo, Spyder, Emacs ou Idle.

**L'environnement Spyder** se décompose comme tous les autres environnements en plusieurs fenêtres.

Il faut bien faire la différence entre les différentes fenêtres, **l'interpréteur** et **l'éditeur** :

- **L'éditeur** sera l'endroit où vous **écrirez** vos programmes en langage Python.
- **L'interpréteur/console** sera l'endroit où vous **exécuterez** vos programmes, où vous taperez vos calculs, où vous pourrez tester l'écriture d'une ligne de commande.
- Je vous conseille de laisser à votre disposition la **fenêtre de l'explorateur de variables** et celle de **l'inspecteur d'objets**.




**ATTENTION : Écrire un programme est différent de l'exécuter ! ! !**

► Écrire un nouveau programme en cliquant sur « fichier » puis « nouveau fichier ».

► Sauvegarder votre programme avec un nom sans espace, sans tiret et sans accents, apostrophes, symboles...

► Une fois votre programme écrit, n'oubliez pas de le sauvegarder avant de l'exécuter (avec F5 ou ▶).

## QUELQUES COMMANDES PYTHON

Rappelons que dans le langage Python les sauts de ligne, les **indentations** les majuscules et les espaces ont leur importance. Voici quelques autres commandes indispensables

▶ La commande `#` sert à écrire des commentaires sur une seule ligne.

▶ Si les commentaires ont lieu sur plusieurs lignes, on les écrit entre `"""..."""` (triples guillemets).

▶ La commande `\` sert pour aller à la ligne dans l'éditeur sans que l'interpréteur lui aille à la ligne.
(*cela sert surtout lorsque nous avons une longue ligne de code et que nous voulons la voir complète à l'écran.*)

▶ La commande `print()` permet d'écrire un texte ou d'**afficher un résultat**.

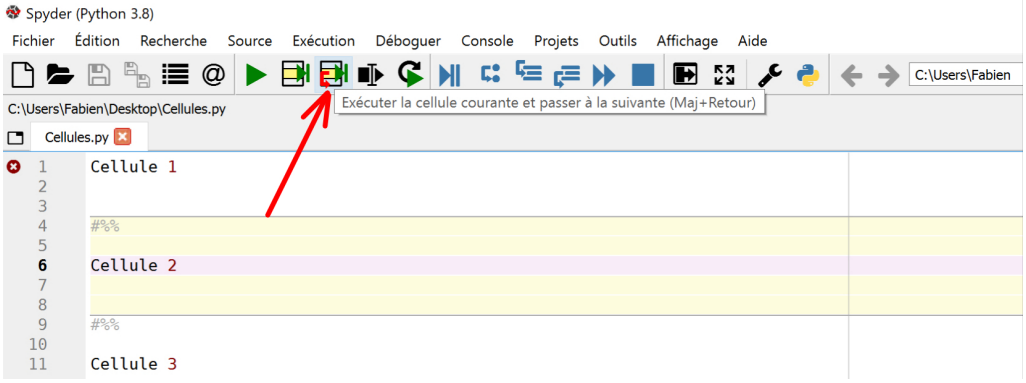
**Attention : par défaut un programme n'affiche pas ce qu'il calcule sans la commande `print()`.**

▶ La commande `input()` permet de demander à l'utilisateur de saisir/définir une variable.
*Attention, si l'on saisit une variable, il faut précéder cette commande du mot clef* `eval()` *afin d'éviter que la variable définie soit interprétée comme une chaîne de caractères.*

▶ Rappelons quelques raccourcis claviers sous Windows :
`CTRL+C` : Copie un morceau de texte préalablement sélectionné (grisé) sous un éditeur.
`CTRL+X` : Coupe un morceau de texte sélectionné, mis dans le « presse-papiers », et peut être ensuite collé.
`CTRL+V` : Colle un morceau de texte qui aura préalablement été mis de côté via un copié ou un coupé.
`CTRL+S` : sauvegarde le fichier sur lequel on travaille sur quasiment tous les éditeurs.
`CTRL+Z` : annule une action sur quasiment tous les éditeurs.
`CTRL+Shift+Z` : annule une annulation ...

## QUELQUES COMMANDES SPYDER COMPLÉMENTAIRES

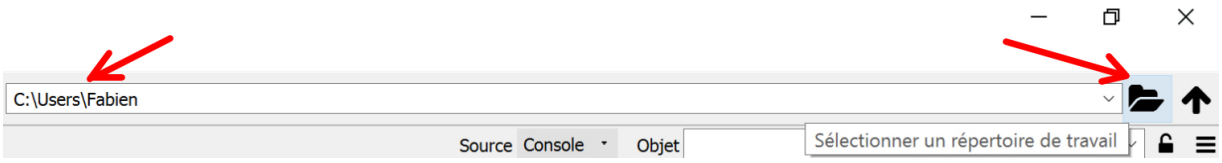
▶ Il est possible de scinder le même fichier en plusieurs **cellules** de sorte à pouvoir les exécuter de façon indépendantes au lieu de ré-exécuter toutes les instructions depuis le début.




*Dans l'exemple ci-contre, le fichier a été séparé en 3 cellules. La cellule active est la cellule 2 (celle où se trouve le curseur) et on clique sur l'onglet indiqué pour n'exécuter que cette cellule.*

▶ Parfois, lors d'écriture de scripts Python on utilise des fichiers annexes pour importer leur contenu, voir même créer des fichiers directement avec Python. Il est nécessaire que les fichiers utilisés soient dans le répertoire de travail indiqué par Spyder.

*Il suffit de le régler sur le bon dossier comme indiqué ci-dessous.*



# VARIABLES ET OPÉRATIONS INFORMATIQUES

## DÉFINITION D'UNE VARIABLE

▶ Une **variable** est une représentation idéale d'une zone de mémoire de l'ordinateur. Il s'agit d'un endroit où l'on peut stocker une valeur, y accéder et changer cette valeur.
Une variable se caractérise par son **nom** (qui permet de l'identifier), une **valeur** (qu'il faut lui affecter) et **type**.

▶ **L'affectation** est l'instruction qui permet d'assigner une valeur à une variable. Si cette dernière n'existe pas encore, on parle de déclaration de la variable.

- ◇ la syntaxe Python dévolue est `variable = valeur.` signifiant `variable`←`valeur`
- ◇ on peut échanger les valeurs de deux variables avec la syntaxe `x,y = y,x`

▶ **Le type** d'une variable correspond à la nature de celle-ci.
Il en existe de nombreux mais parmi eux certains sont à maitriser.

## LES TYPES DE VARIABLES

Voici les différents types de variables qu'il est impératif de connaître :

| **Type de variable** | **Syntaxe Python** | **Description** | Exemple |
|---|---|---|---|
| Entier relatif | `int` | Entiers (64 bits) | `1981` |
| Nombre à virgule | `float` | Nombres décimaux | `3.141592` |
| Liste/Tableau | `list` | Tableaux formés de variables | `[1981,3.14,"PCSI"]` |
| Chaîne de caractères | `str` | Tableaux formés uniquement de lettres | `"Essouriau"` |
| Booléens | `bool` | Peut prendre deux valeurs `True` ou `False` | `True` |
| Tuple | `tuple` | $n$-uplets (couple, triplet) de variables | `(True,"e",3.14)` |
| Dictionnaire | `dict` | collections d'éléments avec clés et valeurs | `{clef:valeur,...}` |

Lorsque l'on définit une variable, il faut donc avoir en tête le type de variable utilisé.

▶ En cas de doute, vous pouvez utilisez `type(x)` si vous voulez connaitre le type de votre variable `x`.

## LES OPÉRATIONS SUR LES VARIABLES

Il est important de connaître les opérations élémentaires que nous pouvons effectuer sur ces variables. Attention, le résultat de ces opérations peut dépendre du type de la variable considérée.

| **Opérateur** | **Utilité** |
|---|---|
| `+` | Addition |
| `-` | Soustraction |
| `*` | Multiplication |
| `/` | Division |
| % | Modulo (*reste de la division euclidienne*) |
| `//` | Quotient(*de la division euclidienne*) |
| `**` | Puissance (*et surtout pas* $\wedge$ !) |
| `+=` | Rajouter à la variable |
| `-=` | Enlever à la variable |
| `*=` | Multiplier à la variable. |
| `/=` | Diviser à la variable |
| `==` et `!=` | égal et différent (test) |
| `<` et `>` | inférieur et supérieur (test) |
| `<=` et `>=` | inférieur ou égal et supérieur ou égal (test) |

# ENTIERS ET FLOTTANTS

## ENTIERS

| Type de variable | Syntaxe Python | Description | Exemple |
|---|---|---|---|
| Entier relatif | `int` | Entiers (64 bits) de -2147483648 à 2147483647 | `1981` |

▶ $[\![-2147483648, 2147483647]\!] = [\![-2^{32}, 2^{32}-1]\!]$, ce qui fait $2^{64}$ nombres relatifs représentables sur 64 bits. Néanmoins tout entier relatif $x$ n'étant pas dans cet intervalle est quand même codé sans problème par Python (pas de dépassement de capacité) sous la forme « **d'entier long** » (*cet entier relatif sera stocké sous la forme d'une collection de nombres représentatifs sous 64 bits qui seront concaténées pour représenter ce nombre $x$*).

▶ La commande `int(x)` transforme le type `float` en type `int` (`int(2.5)` retourne 2)

**Exemple.** Voici quelques exemples :

| Commande dans l'interpréteur | Résultat obtenu |
|---|---|
| `3.1-2.5` | `0.6` |
| `1.5**2.1` | `2.3431044239829237` |
| `13%4` | `1` |
| `13//4` | `3` |
| `x=4` puis `x+=3` | `7` |
| `x=17` puis `x//=3` | `5` |
| `5<4` | `False` |
| `1<=1` | `True` |
| `1+1==2` | `True` |

## FLOTTANTS

| Type de variable | Syntaxe Python | Description | Exemple |
|---|---|---|---|
| Nombre à virgule | `float` | Nombres décimaux | `3.141592` |

▶ Tout nombre réel $x$ est en fait représenté sur 64 bits à l'aide de sa **forme normalisée** (signe, **exposant** et **mantisse**), la taille étant limité à 64 bits, le code associé est donc très souvent seulement une (bonne) approximation du réel $x$.

▶ La commande `float(x)` transforme le type `int` en type `float` (`float(2)` retourne `2.0`)

**Exemple.** Voici quelques exemples :

| Commande dans l'interpréteur | Résultat obtenu |
|---|---|
| `3+3` | `6` |
| `2**3` | `8` |
| `13.0%4` | `1.0` |
| `13/4` | `3.25` |
| `x=4` puis `x*=0.5` | `2.0` |
| `0.1+0.2==0.3` | `False` |

*Le résultat de la dernière ligne du tableau peut paraitre étonnant car mathématiquement l'égalité est vraie et on s'attend à* `True` *mais la représentation de ces deux flottants n'étant qu'approchée, on n'a pas égalité de leur représentation sur 64 bits ce qui explique le résultat* `False`.

▶ C'est la raison pour laquelle **on bannit le test d'égalité entre deux flottants.**

# TABLEAUX ET LISTES

## LISTES

| Type de variable | Syntaxe Python | Description | Exemple |
|---|---|---|---|
| Liste/Tableau | `list` | Tableaux formés de variables | `[1981,3.14,"PCSI"]` |

▶ La commande `list(x)` transforme le type `str` en type `list` (`list("texte")` retourne `["t","e","x","t","e"]`)

▶ La commande `[x]` transforme le type `int` ou `float` en type `list` (par exemple `[3]` ou `[3.0]`)

**Exemple.** Voici quelques exemples :

| Commande dans l'interpréteur | Résultat obtenu |
|---|---|
| `3*[1,2,3]` | `[1,2,3,1,2,3,1,2,3]` |
| `[4,5,6]+[1,2,3]` | `[4,5,6,1,2,3]` |
| `[1,2]==[2,1]` | `False` |
| `[k**2 for k in range(5)]` | `[0,1,4,9,16]` |

▶ **Liste à une dimension :**

◇ Liste vide : `L=[]`

◇ Définir une liste de petite taille par ses éléments : `L=[45,True,-3.0,"texte"]`.

◇ Définir une liste nulle **par répétition** `L=[0]*n` (*liste de taille $n$ remplie de* 0)

◇ Définir une liste de taille $n$ **par compréhension** : `L=[3**i for i in range(n) if i%2==1]`

◇ Longueur d'une liste : `len(L)`

◇ Aller chercher l'élément n°`k` d'une liste : `L[k]`, le remplacer par `x` : `L[k]=x`, le dernier élément : `L[-1]`

**(ATTENTION : l'index des éléments démarre à 0 !!!)**

◇ Ajouter un élément `x` (à la fin) d'une liste `L` : `L.append(x)` (`L+=[x]` à éviter)

◇ Retirer le dernier élément d'une liste `L` : `L.pop()`

◇ Retirer l'élément d'indice `i` d'une liste `L` et renvoie sa valeur : `L.pop(i)` (*hors-programme*)

▶ **Parcourir les éléments d'une liste :**

**Élément par élément**

```
1 L=[2,5,4,7,6,3]
2 for x in L:
3   print(x)
```

**Indice par indice**

```
1 L=[2,5,4,7,6,3]
2 for k in range(len(L)):
3   print(L[k])
```

**Remarque : Ne confondez pas l'élément du tableau `T[i]` avec son indice `i`.**
(*pour la première méthode de parcours d'un tableau, il est impossible de récupérer les indices des éléments.*)

▶ **Tranches dans une liste :**

◇ `L[debut:fin]` renvoie la liste contenant les éléments de `L` de `L[debut]` (inclus) jusqu'à `L[fin]` (exclus).

◇ `L[debut:]` renvoie une liste contenant les éléments de la liste `L` de `L[debut]` (inclus) jusqu'à la fin.

◇ `L[:fin]` renvoie une liste contenant les éléments de la liste `L` du début jusqu'à `L[fin]` (exclus).

◇ `L[:i]+L[i+1:]` créé une nouvelle liste sans l'élément d'indice `i` de `L`.

► **Liste à deux dimensions :**

◇ Une liste à deux dimensions est une liste de listes :

```
L=[[valeur1_1,...],[valeur2_1,...],...]
```

Vous remarquez qu'il y a des crochets à l'intérieur d'autres crochets, ceci veut dire qu'il y a des listes imbriquée, il s'agit donc d'une liste ayant d'autres listes comme valeurs.
Chaque imbrication ajoute une dimension à la liste. (listes à trois dimensions, etc...)

◇ Accéder aux éléments d'une liste de listes : `L[i][j]` (*élément d'indice $j$ de la liste d'indice $i$ de* `L`)
**Exemple :** `L[0][0]` est le premier élément de la première liste
**Exemple :** `L[1][-2]` est l'avant-dernier élément de la seconde liste (liste d'indice 1)
**Exemple :** `L[0]` est la première « élément-liste » de `L` en entier

◇ Parcourir les éléments d'une liste de listes (forcément une double boucle) :

```
L=[[2,5,4,1],[5,8,9],[2],[4,7,8,6,4,7,8],[3,6],[5],[2,6,5,7,4]]
```

**Élément par élément**

```
for l in L:
    for x in l:
        print(x)
```

**Indice par indice**

```
for i in range(len(L)):
    for j in range(len(L[i])):
        print(L[i][j])
```

◇ Si chaque « élément-liste » a la même longueur on peut alors plutôt parler de « matrice ».
Nombre de « lignes » : `n=len(L)` et Nombre de colonnes : `p=len(L[0])`
« Matrice » nulle de taille $n \times p$ : `M=[[0]*p for i in range(n)]`
*Ces matrices servent par exemple pour stocker les valeurs des pixels d'une image (noir et blanc/couleur).*

► **Copie superficielle et en profondeur :**

◇ Si `L` est une liste et si `M=L`, alors une modification de `M` entraîne une modification de `L` (et vice-et-versa).
On parle de **copie superficielle**.
Rappelons enfin qu'une fonction modifie une liste même sans `return`, même s'il cette liste est modifiée en tant que variable locale. C'est ce qu'on appelle **l'effet de bord**.

◇ Si vous voulez faire évoluer `L` et `M` de façon indépendante, la commande `M=L.copy()` pare le problème.
On parle de **copie en profondeur**.

Si jamais vous avez oublié la commande `copy` ce n'est pas grave : `M=L.copy()` ⇔ `M=[l for l in L]` ⇔ `M=L[:]`

## TABLEAUX

► Les **listes** sont des **tableaux**, ce qui les diffèrent, c'est que **les valeurs des listes peuvent être de types différents** (une liste peut contenir un nombre et un texte.)
*Ce qui a été expliqué précédemment sur les listes reste valable pour les tableaux.*

```
liste=["mercredi",19,"septembre",2015]
tableau=["mercredi","19","septembre","2015"]
```

## TUPLES

| Type de variable | Syntaxe Python | Description | Exemple |
|---|---|---|---|
| Tuple | `tuple` | $n$-uplets (couple, triplet) de variables | `(True,"e",3.14)` |

► Les éléments d'un tuple sont **ordonnés** et on accède à un élément grâce à sa position en utilisant un numéro qu'on appelle l'**indice** de l'élément comme une liste.
► Contrairement aux listes, les tuples ne sont **pas modifiables**. *(moins modulables que les listes, moins utilisés)*

# CHAÎNES DE CARACTÈRES

| Type de variable | Syntaxe Python | Description | Exemple |
|---|---|---|---|
| Chaîne de caractères | `str` | Tableaux formés uniquement de lettres | `"Essouriau"` |

▶ Les **chaînes de caractères** sont des variables contenants du texte. Elles se disent de type `str`.
▶ Les chaînes de caractères s'écrivent entre guillemets `"` ou apostrophes `'`.
▶ La plupart des opérations définies pour les tableaux s'appliquent aux chaînes de caractères mais pas toutes.

▶ La commande `str(x)` transforme le type `int`,`float` ou `list` en type `str`.
(`str(3)` retourne `"3"`,`str(3.0)` retourne `"3.0"` et `str([1,2])` retourne `"[1,2]"`)

**Exemple.** Voici quelques exemples :

| Commande dans l'interpréteur | Résultat obtenu |
|---|---|
| `3*"Aie "` | `"Aie Aie Aie "` |
| `"I am"+"clever"` | `"I amclever"` |
| `"ab"=="ba"` | `False` |
| `len("Ca va?")` | `6` |

▶ On remarque que l'espace `" "` compte pour un caractère à part entière ! Quelques autres caractères spéciaux.

| Utilité | Caractère |
|---|---|
| Aller à la ligne | `\n` |
| Afficher un guillemet | `\"` |
| Afficher une apostrophe | `\'` |

| Utilité | Caractère |
|---|---|
| Afficher un anti-slash | `\\` |
| Saut de page | `\f` |

▶ **Quelques commandes sur les chaînes :**
◇ Chaîne vide : `chaine=""`
◇ Définir une chaîne de petite taille par ses éléments : `chaine="Voilà ce que je voulais vous dire!"`.
◇ Longueur d'une chaîne : `len(chaine)`
◇ Élément n°`k` d'une chaîne : `chaine[k]`, le dernier élément : `chaine[-1]` **(l'index commence à 0 !)**
◇ **On ne peut pas modifier une chaine, ajouter/retirer des éléments à une chaine !**
**Exemple :** `chaine.append("x")` renvoie une erreur, comme `chaine[k]="z"` ou encore `chaine.pop()`.
◇ Même si les chaînes de caractères sont **immuables** ou **non modifiables**, pour contourner ce problème on peut effacer l'ancienne chaîne au profit d'une nouvelle contenant les modifications.
**Exemple :** `chaine="Bomjour!"`
Modifier un caractère : `chaine=chaine[:2]+"n"+chaine[3:]`
Supprimer le dernier caractère : `chaine=chaine[:-1]`

▶ **Parcourir les éléments d'une chaîne :**

**Élément par élément**

```
chaine="Bonjour à tous!"
for lettre in chaine:
  print(lettre)
```

**Indice par indice**

```
chaine="Bonjour à tous!"
for k in range(len(chaine)):
  print(chaine[k])
```

**Remarque : Ne confondez pas l'élément de la chaîne `chaine[i]` avec son indice `i`.**
(*pour la première méthode de parcours d'une chaîne, il est impossible de récupérer les indices des éléments.*)

▶ **Tranches dans une chaîne :** identiques aux tranches dans une liste.

▶ **Copie superficielle et en profondeur** Pas de problème pour les chaînes de caractères.

▶ Quelques commandes supplémentaires (à ne pas retenir mais elles existent) :
`chaine.lower()`, `chaine.upper()`, `chaine.find("lettre")`, `chaine.count("lettre")`, `chaine.index("lettre")`

# DICTIONNAIRES

| Type de variable | Syntaxe Python | Description | Exemple |
|---|---|---|---|
| Dictionnaire | `dict` | collections d'éléments avec clés et valeurs | `{clef:valeur,...}` |

▶ Un **dictionnaire** (type `dict`) est une collection d'**éléments** composés d'une **clé** associée à une **valeur**.

▶ Contrairement aux listes qui sont délimitées par des crochets, on utilise des **accolades** pour les dictionnaires.

▶ Un dictionnaire en Python va permettre de rassembler des éléments (comme des listes ou tuples) mais ceux-ci seront **identifiés par une clé**. *Analogie à un dictionnaire où on accède à une définition avec un mot.*

▶ **Commandes utiles pour un dictionnaire :**

**Méthode 1 - Directement par la liste de ses éléments :**
(par exemple le nombre de roues d'un type de véhicule) :

```
dico = {"voiture": 4, "vélo": 2, "tricycle": 3}
```

**Remarque.**
- Ici les clés sont `"voiture"`, `"vélo"` et `"tricycle"`.
- Les valeurs sont les entiers `4`, `2` et `3`.
- Il n'y a **pas d'ordre** entre les clefs ! (clefs non numérotées)
- **Les clefs peuvent être n'importe quel type de variable sauf des listes.**

**Méthode 2 - Ajout un par un des éléments**

```
dico = {}

dico["voiture"]=4
dico["vélo"]=2
dico["tricycle"]=3
```

**Remarque.**
- Si une clé n'existe pas dans le dictionnaire **elle est créé avec la valeur associée**.
- Si une clé existe déjà dans le dictionnaire, **sa valeur est remplacée**.

**Exemple.** Exemple de dictionnaire avec différents types de variables :

```
dico={6:"petit" , "violet":True , False:[1,2,3]}
```

**Obtenir les clés :**

```
for key in dico:
    print(key)
```

**Obtenir les valeurs :**

```
for key in dico:
    print(dico[key])
```

| Commandes | Description |
|---|---|
| `val=dico[key]` | Stocke dans `val` la valeur associée à la clé `key` dans `dico` |
| `dico[key]=val` | Remplace la valeur associée à la clé `key` par `val` dans `dico` (si `key` était déjà présente) |
| `dico[key]=val` | Ajoute la clé `key` et sa valeur associée `val` dans `dico` (si `key` n'était pas présente) |
| `if key in dico:` | Teste si la clé `key` est dans `dico` (*complexité en* $\mathcal{O}(1)$) |
| `if dico[key]==val:` | Teste si la valeur associée à la clé `key` est égale à `val` |

# COMPARATIF LISTES/TUPLES/CHAÎNES/DICTIONNAIRES

Le tableau ci-dessous regroupe les principales spécificités et différences entre ces 4 types de variables.

*On utilisera le plus adapté en fonction de ce qui est demandé.*

| Tableau | Liste | Tuple | Chaine de caractères | Dictionnaire |
|---|---|---|---|---|
| type `list` | type `list` | type `tuple` | type `str` | type `dict` |
| `["a",True,7]` | `[3.14,2.71,1,41]` | `("a",True,7)` | `"Essouriau"` | `{ clé : valeur , ... }` |
| modifiable | modifiable | **non modifiable** | **non modifiable** | modifiable |
| ordonné | ordonné | ordonné | ordonné | **non ordonné** |

# INSTRUCTIONS CONDITIONNELLES

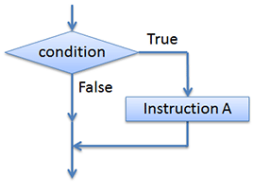
Une **instruction conditionnelle** permet d'exécuter des instructions seulement sous une certaine condition. En Python, on utilise les mots clefs `if`, `else` et `elif` qui consistent à tester (avec des opérateurs de test) si une affirmation est vraie ou non.

## INSTRUCTIONS CONDITIONNELLES

### ▶ L'instruction `if ... :`

◇ Une condition s'effectue avec le mot clé `if ... :` (mot anglais qui veut dire « si »).

```
age=eval(input("Tu as quel âge ?"))

if age==18: # On teste si l'age est égal à 18.
  print("Tu es majeur")
```



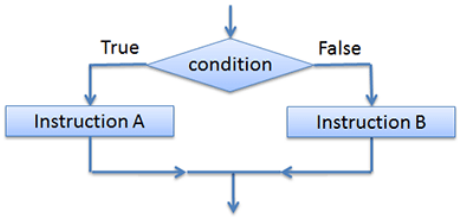

◇ On a besoin d'opérateurs de test pour les conditions, voici les principales :

| **Opérateur** | == | != | < | > | <= | >= | or | and | not |
|---|---|---|---|---|---|---|---|---|---|
| **Teste** | égal | non égal | inférieur | supérieur | inférieur ou égal | supérieur ou égal | ou | et | non |

### ▶ L'instruction `else:`

Le mot clef `else:` permet d'exécuter une instruction lorsque que la condition du `if` n'est pas vérifiée. (« else » signifie « sinon » en anglais.)

```
age=eval(input("Tu as quel âge ?"))

if age==18:
  print("Tu es majeur.")
else:
  print("Tu n'as pas 18 ans.")
```




### ▶ L'instruction `elif ... :`

◇ Le mot clef `elif:` permet d'exécuter une instruction lorsque que la condition du `if` n'est pas vérifiée et que celle du `elif` est vérifiée. (« elif », pour « else if » soit « sinon si ».)

◇ Permet d'éviter de multiplier les tests imbriqués avec `if`.

```
age=eval(input("Tu as quel âge ?"))

if age==18:
  print("Tu es majeur")
elif age==17:
  print("Encore qq jours et tu seras majeur.")
else:
  print("Tu n'as pas 18, ni 17 ans.")
```

### ▶ L'instruction `if ... in ...`

Pour chercher un caractère/un mot dans un « texte »/une chaîne de caractère, on utilise `if ... in ... :` (traduire par « si ... est dans ... »). On peut également chercher si un élément est dans une liste, un dictionnaire ou un tuple.

```
Texte="J'aime être en PCSI à l'Essouriau"
if 'PCSI' in Texte:
  print("Le mot recherché est présente dans le texte demandé.")
else:
  print("Le mot recherché n'est pas présent dans le texte demandé.")
```

▶ Instructions imbriquées

Attention au niveau d'indentation lorsque l'on utilise plusieurs boucles imbriquées. Par exemple :

```
age=eval(input("Tu as quel âge ?"))

if age<18:
  print("Tu es mineur.")
  if age<=3:
    print("Tu es un bébé.")
else:
  print("Tu es majeur.")
```

```
age=eval(input("Tu as quel âge ?"))

if age<18:
  print("Tu es mineur.")
  if age<=3:
    print("Tu es un bébé.")
  else:
    print("Tu es majeur.")
```

▶ Astuces

◇ Utiliser `and` ou `or` permet d'éviter plusieurs `if` ! par exemple pour $x$ entier, $1 \leqslant x \leqslant 100$ et impair on a :

```
if 0<x and x<=100 and x%2==1:
```

◇ Si une variable `Test` est un booléen `True` ou `False`, inutile d'utiliser un opérateur d'égalité dans un test. On peut directement écrire :

```
if Test:
  print("C'est vrai!")
```

◇ La valeur du booléen `True` est reconnue par Python comme égale à celle de l'entier `1` !
Si on fait `True==1` on obtient étonnamment `True`. De même `False==0` donne `True`.

# INSTRUCTIONS ITÉRATIVES

Les **structures de contrôle itératives** ou **boucles** permettent d'exécuter plusieurs fois de suite une ou plusieurs instructions. Les deux types à maitriser sont : `for` et `while`.

## INSTRUCTION ITÉRATIVE `for ... in ... :`

Les boucles inconditionnelles servent à exécuter une instruction un **nombre connu de fois**.

▶ L'instruction `for k in range(debut,fin,pas):`

◇ `range(a,b,h)` s'utilise de la façon suivante :

— `a` est la valeur initiale que le compteur `i` va prendre. Par défaut si elle n'apparait pas, elle vaut 0.

— `b-1` est la limite où la condition s'arrête.

— `h` est **le pas** avec lequel `i` augmente (diminue). Par défaut si elle n'apparait pas le pas vaut +1.

◇ Exemple pour imprimer 10 lignes :

```
for i in range(1,11):
  print("J'imprime la ligne numéro ",i,".")
```

◇ Exemples : `range(12)` donne dans l'ordre les entiers $\{0,1,2,3,4,5,6,7,8,9,10,11\}$, `range(2,12,2)` donne $\{2,4,6,8,10\}$ et `range(9,2,-1)` donne dans l'ordre $\{9,8,7,6,5,4,3\}$.

▶ L'instruction `for k in liste:` ou `for k in chaine:`

Si vous devez parcourir du texte (chaîne de caractères), vous pouvez utilisez `for ... in ... :` , par exemple :

```
prenom=input("Entrer votre prénom.")
for i in prenom:
  if i=="a":
    print("Votre prénom contient la lettre a.")
```

## INSTRUCTION ITÉRATIVE `while ... :`

Les boucles conditionnelles peuvent servir à exécuter une instruction un nombre de fois dépendant de la réalisation d'une certaine condition.

◇ On utilise le mot clé `while ...` : qui peut être traduit par « tant que ... ».

◇ Voici un exemple analogue à celui avec `for` :

```
i=1
while i<=10 :
  print("Je recopie 10 fois la même ligne.")
  i+=1
```

◇ Notez la nécessité d'une variable aléatoire « muette » `i` qui s'appelle un **compteur**. Elle représente en général le numéro de la boucle que l'on est en train de parcourir et intervient parfois dans le test d'arrêt. Il est nécessaire de **l'incrémenter** (`i+=1`) à chaque boucle afin de pouvoir compter le nombre de boucles.

◇ Un autre exemple où `while` est incontournable cette fois, sans nécessité d'avoir un compteur :

```
u0=2000
while u0>0 :
  print("Le terme de la suite vaut ",u0)
  u0=np.log(u0)
```

# FONCTIONS PYTHON

## FONCTION

▶ Pour optimiser un algorithme, on va **remplacer des bouts de codes qui se répètent par une fonction**.
*Ainsi, en apprenant à faire des fonctions, vous rendrez votre code plus lisible.*

▶ Quasiment toutes les questions des sujets de concours demandent d'écrire ou d'examiner des fonctions.

▶ En programmation, les **fonctions** sont identiques aux fonctions mathématiques.
À une valeur `x` (ou plusieurs), la fonction va retourner une valeur `y` (ou plusieurs).
Une **fonction** en informatique est donc une séquence finie d'instructions, dépendant de paramètres d'entrée (appelés **arguments**), et retournant un **résultat (renvoi)**.

```
def fonction(argument):
  intructions
  return resultat
```

◇ Le mot clé `def` pour définir une fonction, puis on nomme la fonction et on termine par `():`.

◇ Dans la parenthèse, on a le choix entre mettre une/des variable(s) d'entrée ou aucune !

◇ Tout ce qui appartient à la fonction sera décalé avec une tabulation.

◇ La fonction se termine par `return` suivi de ce que l'on souhaite retourner.
**Noter que lors de l'exécution de l'instruction `return` la fonction obligatoirement prend fin.**

```
# 1) Je définis ma fonction.
# (Fonction lue mais ne pas exécutée!)
def carre(x):
  y=x**2
  return y
```

```
# 2) J'exécute la fonction (entrée = 3).
resultat=carre(3)

# 3) J'affiche le résultat renvoyé.
print(resultat)
```

▶ Entrées et sorties multiples.
*Voici un exemple qui permet de faire connaissance avec une affectation multiple (triple ici) :*

```
def exemple(x,y,z):
  w1,w2,w3=x+y+z,x*y+y*z+x*z,x*y*z
  return w1,w2,w3

x,y,z=exemple(1,2,3)
```

## PROCÉDURE

▶ Les **procédures** sont des fonctions sans `return` car elles n'ont pas besoin de renvoyer quelque chose.
En fait, elles ne font que réaliser des actions, elles s'arrêtent quand elles les ont toutes accomplies.

▶ Une procédure ne renvoie rien, donc lors d'un affichage il s'affiche `None`.

## VARIABLES LOCALES/GLOBALES

▶ Lors de la création d'une fonction Python on est amené à utiliser des variables autres que les entrées pour calculer la sortie attendue. Dans ce cas le programme reconnait ces variables dans la fonction mais pas en dehors.

**Ces variables sont dites locales.** Si vous voulez utiliser une variable définie dans une fonction après exécution, il faut le stipuler au début de la fonction à l'aide du mot clef `global`. (*commande hors-programme et possible que pour une variable définie dans une fonction qui n'est pas en entrée.*)

*Dans cet exemple, si la variable `c` est utilisée dans la fonction, alors le programme reconnaitra la valeur qui lui a été associée même en dehors de la fonction.*

```
def fonction(argument):
  global c
  intructions
  return resultat
```

# ALGORITHMES CLASSIQUES

### ▶ Calcul de la somme des éléments d'une liste

```
def somme(L):
  s=0
  for k in L:
    s+=k
  return s
```

```
def somme(L):
  s=0
  for k in range(len(L)):
    s+=L[k]
  return s
```

### ▶ Calcul du produit des éléments d'une liste

```
def produit(L):
  p=1
  for k in L:
    p*=k
  return p
```

```
def produit(L):
  p=1
  for k in range(len(L)):
    p*=L[k]
  return p
```

---

### ▶ Détermination du max/min

```
def maximum(L):
  M=L[0]
  for k in L:
    if k>M:
      M=k
  return M
```

```
def minimum(L):
  m=L[0]
  for k in L:
    if k<m:
      m=k
  return m
```

### ▶ Détermination de l'indice du max/min

```
def maximum(L):
  M=L[0]
  imax=0
  for i in range(len(L)):
    if L[i]>M:
      M=L[i]
      imax=i
  return imax
```

```
def minimum(L):
  m=L[0]
  imin=0
  for i in range(len(L)):
    if L[i]<m:
      m=L[i]
      imin=i
  return imin
```

---

### ▶ Calcul de la moyenne des éléments d'une liste

```
def moyenne(L):
  s=0
  for k in L:
    s+=k
  m=s/len(L)
  return m
```

### ▶ Calcul de l'écart-type des éléments d'une liste

```
def Ecart-Type(L):
  m=moyenne(L)
  e=0
  for k in L:
    e+=(k-m)**2
  return e**0.5
```

---

### ▶ Présence d'un élément dans une liste/chaîne

```
def Presence(L,x):
  for l in L:
    if l==x:
      return True
  return False
```

### ▶ Présence d'un élément dans une liste/chaîne

```
def Presence(L,x):
  for k in range(len(L)):
    if L[k]==x:
      return True
  return False
```

---

### ▶ Nombre d'occurrences d'un élément

```
def Occurrence(L,x):
  occ=0
  for l in L:
    if l==x:
      occ+=1
  return occ
```

### ▶ Nombre d'occurrences des éléments (*dictionnaire*)

**Naïf (quadratique)**

```
def Occurences(L):
  dico={}
  for x in L:
    dico[x]=Occurrence(L,x)
  return dico
```

**Optimale (linéaire)**

```
def Occurences(L):
  dico={}
  for x in L:
    if x in dico:
      dico[x]+=1
    else:
      dico[x]=1
  return dico
```

*Soit $n$ la longueur de la liste. Pour la version naïve, la complexité est quadratique ($\mathcal{O}(n^2)$), car on a une boucle de longueur $n$ dans laquelle à chaque étape utilise le programme* `Occurrence` *qui a aussi une complexité linéaire. Pour la version optimale, on a uniquement une boucle de longueur $n$, chaque étape ne comportant que $\mathcal{O}(1)$ calculs donc la complexité est linéaire ($\mathcal{O}(n)$). L'idée est d'ajouter 1 au nombre d'occurrences de chaque élément dès que l'on l'aperçoit dans la liste, le type dictionnaire permet de bien gérer cela.*

# AUTRES ALGORITHMES ASSEZ CLASSIQUES

### ▶ Liste des indices des éléments égaux au maximum dans un tableau

```
def Maximum(T):
    M=T[0]
    for t in T:
        if t>M:
            M=t
    return M
```

```
def Liste_Indices_Max(T):
    liste_indices=[]
    M=Maximum(T)
    for k in range(len(T)):
        if T[k]==M:
            liste_indices.append(k)
    return(liste_indices)
```

### ▶ Recherche du 2nd maximum dans un tableau (sans le modifier)

```
def Second_Max(T):
    M=Maximum(T)
    debut=0
    while T[debut]==M:
        debut+=1
    M2=T[debut]
    for k in range(debut,len(T)):
        if T[k]>M2 and T[k]!=M:
            M2=T[k]
    return M2
```

### ▶ Recherche des deux valeurs les plus proches dans un tableau

```
def Deux_plus_proches_valeurs(L):
    d=abs(L[0]-L[1])
    ind1,ind2=0,1
    for i in range(len(L)):
        for j in range(i):
            e=abs(L[i]-L[j])
            if e<d:
                d=e
                ind1,ind2=i,j
    return [(L[ind1],L[ind2])
```

### ▶ Calcul des termes d'une suite récurrente

$u_{n+2} = 3u_{n+1} - 2u_n$ avec $u_0 = 4$ et $u_1 = -5$

```
def u(n):
    u0=4
    u1=-5
    if n==0:
        return u0
    if n==1:
        return u1
    for k in range(2,n+1):
        u2=3*u1-2*u0
        u0=u1
        u1=u2
    return u2
```

### ▶ Dépassement d'un seuil $s$ pour une quantité

Par exemple trouver $n$ tel que $u_n \geqslant s$.

```
def seuil(u,s):
  n=0
  while u(n)<s:
    n+=1
  return n
```

### ▶ Moyenne des éléments d'une liste à deux dimensions

```
def moyenne_2D(L):
  S=0
  n=0
  for l in L:
    n+=len(l)
    for x in l:
      S+=x
  return S/n
```

```
def moyenne_2D(L):
  S=0
  n=0
  for i in range(len(L)):
    n+=len(L[i])
    for j in range(len(L[i])):
        S+=L[i][j]
  return S/n
```

# BIBLIOTHÈQUES ET MODULES

Voici certaines **bibliothèques/modules** de Python. Aucune connaissance n'est exigible (toute commande sera rappelé) mais il ne faut pas les découvrir aux concours. *En particulier, lors de l'épreuve oral Maths 2 de Centrale.*

Afin d'utiliser une **bibliothèque** ou **module** dans Python il faut au préalable **l'importer** dans votre environnement Spyder. Il y a plusieurs façons de faire, exemple avec la bibliothèque `math` :

- **Importation d'une fonction particulière.** `from math import sin` par exemple, permet d'importer spécifiquement la fonction `sin`, (ou `cos`, ou la constante `e`, sous ces noms là).
- **Importation de toutes les fonctions.** `from math import *` est similaire à l'importation précédente, mais le joker `*` est utilisé à la place des noms explicites des fonctions.
- **Importation de la bibliothèque.** `import math` importe la bibliothèque, les fonctions sont alors accessibles par `math.sin`, `math.cos`, par exemple.
- **Importation de la bibliothèque et alias.** On abrège le nom de la bibliothèque avec un alias. Avec `import math as m` les fonctions sont alors accessibles par `m.sin`, `m.cos`...

La documentation Python est assez bien fournie, pour y accéder il suffit d'écrire `help(nom_du_module)`. Ceci fonctionne également avec les alias. Par exemple, `import math as m` ; `help(m)` fournira de l'aide sur toutes les fonctions du module. Bien sûr, il est possible d'obtenir de l'aide sur une fonction spécifique comme `m.sqrt`.

- Module `math`
  Ce module contient les fonctions mathématiques usuelles, ainsi que les constantes $e$ et $\pi$.
- Module `matplotlib`
  On utilisera principalement le sous-module `pyplot`, qui sert à tracer des courbes. On l'importera comme suit : `import matplotlib.pyplot as plt`.
- Module `numpy`
  On importera ce module avec l'alias np : `import numpy as np`. Ce module permet la construction de tableaux, listes, matrices et récupération de données propres aux listes (taille, nombre d'éléments...)
  - ◇ Sous-module `linalg`
    Le sous-module `numpy.linalg` (importé comme `import numpy.linalg as alg`). Ce sous-module permet de faire de l'algèbre linéaire (espaces vectoriels et leurs applications) du programme de PCSI et PSI que vous pourrez découvrir ultérieurement. Il permet déjà d'effectuer les opérations matricielles.
  - ◇ Sous-module `random`
    Le sous-module `random` permet de générer des nombres aléatoires, et plus généralement de traiter de probabilités. On l'importe ici comme `import numpy.random as rd`.
- Module `time`
  Permet d'importer la fonction `time()` qui relève le temps depuis une date donnée.
- Module `scipy` (Sous-modules `optimize`, `integrate`, `odeint`)
  Le module `scipy` est le module de calcul scientifique. Trois points au programme : résolution d'équations numériques, intégration de fonctions et résolution d'équations différentielles.
  - ◇ On utilise le sous-module `optimize` de `scipy` : `import scipy.optimize as sco`. La fonction `fsolve` est tout indiquée pour résoudre une équation de la forme $f(x) = 0$.
  - ◇ On utilise le sous-module `integrate` de `scipy` : import `scipy.integrate as sci`. La fonction que l'on va utiliser est `quad`. Il suffit d'indiquer la fonction à intégrer et les bornes pour calculer la valeur de l'intégrale.
  - ◇ L'intégration d'équations différentielles se fait avec le même sous-module, et la fonction `odeint` (pour *ordinary differential equations integration*).
- Autres modules : `copy`, `fractions`, `polynomial`, `itertools`, `tkinter`...

# BIBLIOTHÈQUE `matplolib.pyplot`

Le programme de CPGE est clair : aucune des commandes ci-dessous n'est exigible par les élèves de CPGE. Elle seront toutes rappelées lors des concours.

On se contente ici d'exhiber 3 usages possibles de cette bibliothèque, avec le sous-module `pyplot`.

## TRACÉ D'UN NUAGE DE POINTS

Exemple de syntaxe possible pour tracer le nuage de points $A(-2,3)$, $B(-1,0)$, $C(3,1)$ et $D(5,2)$.

On définit la liste des abscisses et la liste des ordonnées puis on effectue les commandes de tracé avec `plt.plot(X,Y,"x")`.

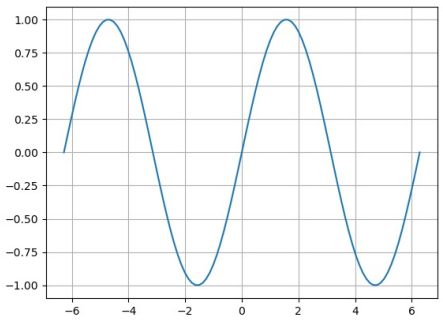
Le `"x"` est une option faisant apparaitre les points avec un symbole de croix (on peut le changer en `"o"`, `"."`, `'v'`....)

```
 # j'importe les packages utiles
import matplotlib.pyplot as plt

# Définition d'une liste d'abscisses et d'ordonnées
X=[-2,-1,3,5]
Y=[3,0,1,-2]
plt.plot(X,Y,"x") # Tracé du nuage de points
plt.grid() # j'affiche le quadrillage
plt.show() # j'affiche le tracé
```

## TRACÉ D'UNE FONCTION

Pour tracer une courbe, définir un liste de points `X` discrétisant l'intervalle $[a,b]$ de définition de $f$ et définir la liste des images `Y`.

Pour représenter sin sur $[-2\pi,\pi]$ ont peut écrire :



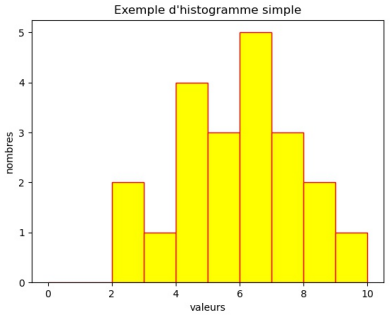

```
import numpy as np # j'importe les bibliothèques
import matplotlib.pyplot as plt

def f(x): # je définis ma fonction
  return np.sin(x)

a=-2*np.pi #je définis l'intervalle [a,b]
b=2*np.pi
n=100 # je définis le nombre de points
# je définis ma liste d'abscisses et d'ordonnées
X=[a+i*(b-a)/n for i in range(n+1)]
# je définis ma liste d'ordonnées
Y=[f(x) for x in X]

plt.plot(X,Y) # je réalise le tracé de f
plt.grid() # j'affiche le quadrillage
plt.show() # j'affiche le tracé
```

- Il existe aussi le module `pylab` combine la fonctionnalité `pyplot` (pour le tracé) avec `numpy`.

## HISTOGRAMME

La liste de commandes ci-dessous donne l'histogramme ci-contre :
(histogramme entre 0 et 10 -`range=(0,10)`- avec 10 intervalles `bins=10`).

```
import matplotlib.pyplot as pl
x = [2,2,3,4,4,4,4,5,5,5,6,6,6,6,6,7,7,7,8,8,9]
pl.hist(x,range=(0,10),bins=10,color='yellow',edgecolor='red')
pl.xlabel('valeurs')
pl.ylabel('nombres')
pl.title('Exemple d\'histogramme simple')
```




- Possibilité de créer des diagrammes en bâton (boite à moustaches), circulaires (camembert) et bien d'autres !

# ALGORITHMES DICHOTOMIQUES

Le programme de CPGE est clair :
• Exponentiation rapide.
• Recherche dichotomique dans un tableau trié.
• *On met en évidence une accélération entre complexité linéaire d'un algorithme naïf et complexité logarithmique d'un algorithme dichotomique. On met en œuvre des jeux de tests, des outils de validation.*

## MÉTHODE DU « DIVISER POUR RÉGNER »

▶ Les algorithmes **dichotomiques** qui utilisent le principe du « Diviser pour régner » afin de diminuer la complexité temporelle (en calcul) d'algorithmes.
▶ L'idée est de passer de la résolution d'un problème de « taille » $n$ à un (ou plusieurs) problème de « taille » $\frac{n}{2}$ plus rapide à résoudre et d'itérer ce principe. (passer à des problèmes de taille $\frac{n}{4}$, puis $\frac{n}{8}$, ...)

Bien entendu, on espère que la résolution de l'ensemble de ces sous-problèmes de taille « petite » soit plus rapide que le problème initial.

## EXPONENTIATION RAPIDE

▶ Soit $a$ un réel et $n$ un entier naturel. Sous Python, la commande `a**n` renvoie la valeur du nombre $a^n$.

On pourrait penser que cette valeur est calculée par Python avec le principe que :

$$a^n = \underbrace{a \times \ldots \times a}_{n \text{ fois}} \text{ ainsi :}$$

```
def Exp(a,n):
    return a**n
```

serait équivalent à

```
def Exp(a,n):
    y=1
    for k in range(1,n+1):
        y*=a
    return y
```

Mais il n'en est rien ! Le calcul de $a^n$ s'effectue à l'aide de **l'exponentiation rapide** :
— Par exemple $a^{16} = (((a^2)^2)^2)^2$. Soit 4 calculs au lieu de 16.
— Si $n$ n'est pas une puissance de 2 ce n'est pas grave : $a^7 = a^4.a^2.a^1 = (a^2)^2.a^2.a = ((a^2).a)^2.a$.
Soit 4 calculs à nouveau au lieu de 7.

▶ L'algorithme d'**exponentiation rapide** repose sur le principe suivant : $a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair} \\ \left(a^{\frac{n-1}{2}}\right)^2 * a & \text{si } n \text{ est impair} \end{cases}$

(*ou encore il se repose sur l'écriture en base* 2 *de l'exposant* $n$)

Qui se traduit en Python par :

**Fonction récursive**

```
def Exp(a,n):
    if n==1:
        return a
    if n%2==0:
        y=Exp(a,n//2)
        return y*y
    else:
        y=Exp(a,(n-1)//2)
        return y*y*a
```

**Fonction itérative**

```
def Exp(a,n):
  y,z,b=a,1,n
  while m>0:
    q=m//2
    r=m%2
    if r==1:
      z*=y
    y*=y
    m=q
  return z
```

▶ L'algorithme d'**exponentiation rapide** est donc basé sur le principe de dichotomie.
Pour un exposant entre $2^p$ et $2^{p+1}-1$ on aura au maximum $p$ étapes, c'est à dire que pour un exposant de $n$, il y a $\frac{\ln(n)}{\ln(2)}$ étapes, soit une complexité asymptotique logarithmique en $O(\ln(n))$.

▶ L'algorithme d'exponentiation « basique » quand à lui a une complexité linéaire en $O(n)$.

## RECHERCHE D'UN ÉLÉMENT DANS UNE LISTE TRIÉE

►**Principe de la recherche par dichotomie** d'un élément $x$ dans une **liste triée** par ordre croissant :

- Si la liste est vide, renvoyer `False`.
- Définir la plage d'indices de recherche notée `[g,d]` (pour gauche et droite) qui au départ vaut `[0,len(L)-1]`.
- Tant que `g` sera inférieur ou égal à `d`, faire :
  - le calcul de l'indice milieu `m` : $m = \left\lfloor \frac{g+d}{2} \right\rfloor$
  - Si `x=L[m]`, alors on a trouvé l'élément `x` dans `L`.
  - Si `x>L[m]`, alors la plage d'indices de recherche `[g,d]` devient `[m+1,d]` (`x` est dans la plage de droite).
  - Sinon la plage d'indices de recherche `[g,d]` devient `[g,m-1]` (`x` est dans la plage de gauche).
- Renvoyer `True` ou `False` selon le fait que `L[m]` soit égal à `x` ou pas pour le dernier `m` calculé.

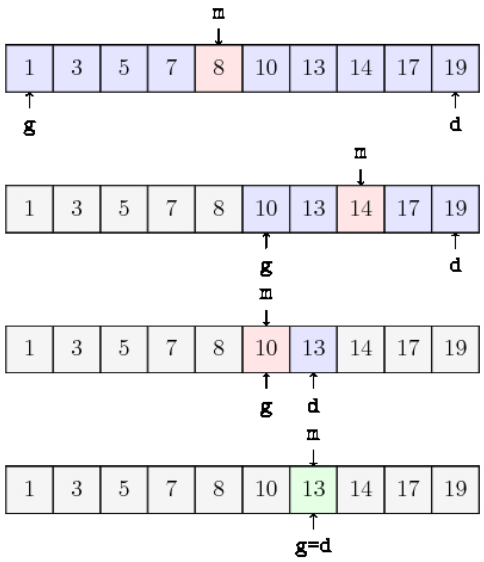
```
def recherche_dicho(x,L):
    if len(L)==0:
        return False
    g = 0
    d = len(L)-1
    while g <= d:
        m = (g+d)//2
        if L[m] == x:
            return True
        elif L[m] < x:
            g = m+1
        else:
            d = m-1
    return False
```

Pour une liste de $n$ éléments, au maximum y a $p$ étapes et :

$$\frac{n}{2^p} \approx 1 \Leftrightarrow \ln(n) - p\ln(2) \approx 0 \Leftrightarrow p \approx \frac{\ln(n)}{\ln(2)}$$

► Complexité asymptotique logarithmique en $O(\ln(n))$.

On cherche si 13 appartient à la liste triée :




Dans cet exemple, avec une liste de 10 éléments `L=[1,3,5,7,8,13,14,17,19]` avec `x=13` :
Étape 1 : `g=0` et `d=9` donc `m=4`. Comme `13>L[4]` alors `g=5` et `d=9`
Étape 2 : `g=5` et `d=9` donc `m=7`. Comme `13<L[7]` alors `g=5` et `d=6`
Étape 3 : `g=5` et `d=6` donc `m=5`. Comme `13>L[5]` alors `g=6` et `d=6`
Étape 4 : `g=6` et `d=6` donc `m=6`. Comme `13=L[6]` alors `x=13` est bien dans la liste `L`

# ALGORITHMES RÉCURSIFS

Le programme de CPGE est clair :
- Version récursive d'algorithmes dichotomiques.
- Fonctions produisant à l'aide de `print` successifs des figures alphanumériques.
- Dessins de fractales.
- Énumération des sous-listes ou des permutations d'une liste.
- *On évite de se cantonner à des fonctions mathématiques (factorielle, suites récurrentes).*
- *On peut montrer le phénomène de dépassement de la taille de la pile.*

## PRINCIPE DES ALGORITHMES RÉCURSIFS

▶ Les **algorithmes récursifs** sont des algorithmes qui résolvent un problème en calculant des solutions d'instances plus petites du même problème. On aura des fonctions qui s'appellent elle-même lors de leur exécution.

L'approche **récursive** est un des concepts de base en informatique, complémentaire de l'approche **itérative** (avec des structures de boucles). *Comme les algorithmes récursifs s'appellent eux-même, on rencontre des problèmes qui leur sont propres.*

## CALCUL DE $n!$ EN ITÉRATIF ET RÉCURSIF

Tout algorithme peut être écrit sous une forme **itérative** ou **récursive**, exemple avec la factorielle :

**Version itérative**

```
def factorielle(n):
  f=1
  for k in range(1,n+1):
    f*=k
  return f
```

**Version récursive**

```
def factorielle(n):
  if n==0:
    return 1
  return n*factorielle(n-1)
```

*On s'aperçoit que la fonction* `factorielle` *à l'étape $n$ s'appelle elle-même à l'étape $n-1$, d'où la nécessité de d'initialiser $0!=1$.*

## CALCUL DE TERMES D'UNE SUITE

**▶ Calcul itératif d'une suite**

$u_{n+1}=\frac{1}{2}u_n-1$ avec $u_0=3$

```
def u(n):
    u=3
    for k in range(1,n+1):
        u=u/2-1
    return u
```

**▶ Calcul récursif d'une suite**

$u_{n+1}=\frac{1}{2}u_n-1$ avec $u_0=3$

```
def u(n):
    if n==0:
        return 3
    return u(n-1)/2-1
```

Le calcul de `u(5000)` avec la version récursive renvoie `RecursionError: maximum recursion depth exceeded in comparison` : on a dépassé la limite de la **pile d'exécution** c'est à dire le nombre maximal d'appels récursif autorisé par Python (*s'obtient avec* `sys.getrecursionlimit()` *de la bibliothèque* `sys` *et est modifiable*).

| Avantages | Inconvénients |
|---|---|
| • Programmation claire<br>• Simple à comprendre | • Complexité très sensible à la programmation évoluant rapidement en puissance (ex : $a^n$), $a$ nombre d'auto-appels récursifs à chaque étape de complexité $O(1)$<br>• Limite de la taille de la pile d'exécution<br>• Preuve et terminaison de l'algorithme plus complexes à montrer<br>• Création de nouvelles variables locales à chaque exécution (mémoire)<br>• Les variables étant locales, pour récupérer des infos sur l'exécution, il faut déclarer des variables globales ou les mettre en argument (cf compteurs) |

## COEFFICIENTS BINOMIAUX

Le but est de calculer les coefficients binomiaux à l'aide de la formule du triangle de Pascal :

$$\forall n \in \mathbb{N}, \forall p \in [\![0, n]\!], \quad \binom{n+1}{p} = \binom{n}{p} + \binom{n}{p-1}$$

Une fonction récursive possible est :

```
def binome(n,p):
    if p==0 or p==n:
        return 1
    return binome(n-1,p)+binome(n-1,p-1)
```

**Remarque :** *Là encore on voit la dangerosité de la récursivité car dans le pire des cas le nombre d'appels double pour passer de l'étape $n$ à l'étape $n-1$ et donc la complexité asymptotique est* ***exponentielle*** *en $O(2^n)$.*

## ÉNUMÉRATION DE SOUS-LISTES DANS UNE LISTE

▶ Soit une liste `L` d'entiers distincts. **On veut lister toutes les sous-listes de `L` avec $k$ éléments.** Soit une liste `L=[1,2,3,4]` et $k = 2$. Nous pouvons attendre deux types de résultats :

— V1 : Les sous-listes sans les permutations possibles : `[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]`

— V2 : Les sous-listes avec toutes les permutations possibles : `[[1,2],[2,1],[1,3],[3,1],...]`

▶ Il y a $A_n^k = \frac{n!}{(n-k)!}$ de sous-listes à $k$ éléments parmi $n$ éléments de type V1.
Il y a $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ de sous-listes à $k$ éléments parmi $n$ éléments de type V2.

▶ Le principe de l'algorithme consiste à traiter le cas de base lorsque $k = 1$ puis à créer par récursivité l'énumération de plusieurs sous-listes de `L` en $k-1$ éléments et de leur ajouter ce qui leur manque.
On ajoute alors tous les résultats attendus en $k$ éléments dans une liste qui est renvoyée.

**• Pour les listes de type V1 :**

```
def Enum_T(L,k):
    if k==1:
        return [[l] for l in L]
    sousL=[]
    for l in Enum_T(L,k-1):
        for elt in L:
            if elt not in l:
                ajout=l+[elt]
                sousL.append(ajout)
    return sousL
```

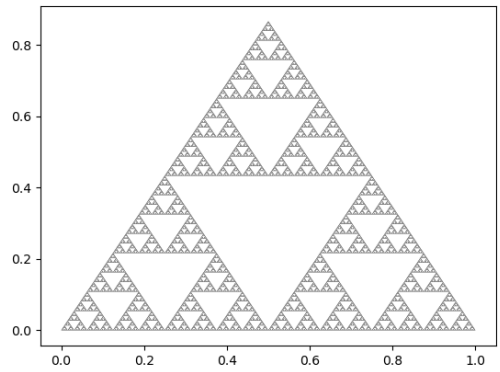
**• Pour les listes de type V2**
(on retire les listes mal ordonnées à chaque étape) :

```
def Enum_U(L,k):
    if k==1:
        return [[l] for l in L]
    sousL=[]
    for l in Enum_T(L,k-1):
        for elt in L:
            if elt not in l:
              if L.index(elt)>L.index(l[-1]):
                ajout=l+[elt]
                sousL.append(ajout)
    sousL2=[]
    for sL in sousL:
        test=True
        for i in range(len(sL)-1):
            if L.index(sL[i])>L.index(sL[i+1]):
                test=False
        if test:
            sousL2.append(sL)
    return sousL2
```

## TRACÉE DE FRACTALES

On veut tracer la **figure fractale** (*objet mathématique qui présente une structure similaire à toutes les échelles*) appelée **triangle de Sierpinski**.

La figure finale ressemble à celle ci-dessous :




La fonction `Polygon` ci-contre permet de tracer une sur une figure un polygone donc on définit les $n$ sommets en argument sous la forme d'une liste de coordonnées `[[`$X_A$`,`$Y_A$`],[`$X_B$`,`$Y_B$`],...,[`$X_{...}$`,`$Y_{...}$`]]` de la couleur voulue.

L'exécution de la commande `Polygon([A,B,C],'black')` fait afficher à l'écran la figure de gauche ci-dessous ($n = 0$).

On écrit ensuite une fonction `Sierpinski(n,A,B,C)` qui superpose sur ce triangle noir un triangle blanc de sommets adaptés pour obtenir la figure à l'étape $n = 1$, pour chaque petit triangle noir va à chaque fois superposer à nouveau le triangle blanc « au centre », étape par étape pour obtenir la fractale voulue (en s'arrêtant à la $n$-ième l'étape).

Il faut donc que la fonction `Sierpinski` s'appelle 3 fois à chaque étape !...

Pour finir on exécute la fonction une unique fois pour faire le tracé (sans `print`).

```
#importation du module de tracé graphique
from matplotlib import pyplot as plt
#fermeture des fenêtres graphiques ouvertes
plt.close('all')

def Polygon(Liste_Points,couleur) :
  #fonction de tracé de polygone "plein"
    X = []
    Y = []
    for Point in Liste_Points :
      #liste des abscisses des points
        X.append(Point[0])
        #liste des ordonnées des points
        Y.append(Point[1])
        #remplissage polygone avec couleur
        plt.fill(X,Y,color=couleur)

import numpy as np
#Sommets d'un triangle équilatéral
A=[0,0]
B=[0.5,np.sqrt(3)/2]
C=[1,0]

#tracé du triangle en noir
Polygon([A,B,C],'black')

def Sierpinski(n,A,B,C):
    if n==0:
        Polygon([A,B,C],'black')
    if n>0:
        I=[(A[k]+B[k])/2 for k in range(2)]
        J=[(B[k]+C[k])/2 for k in range(2)]
        K=[(C[k]+A[k])/2 for k in range(2)]
        Polygon([I,J,K],'white')
        Sierpinski(n-1,A,I,K)
        Sierpinski(n-1,I,B,J)
        Sierpinski(n-1,K,J,C)

Sierpinski(n,A,B,C)
```

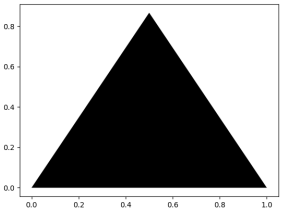


Figure pour $n = 0$

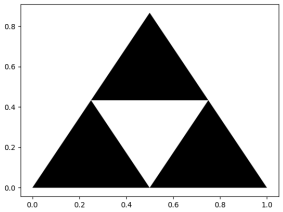


Figure pour $n = 1$

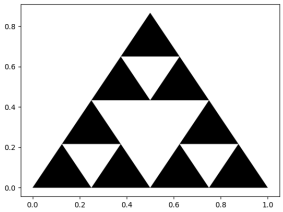


Figure pour $n = 2$

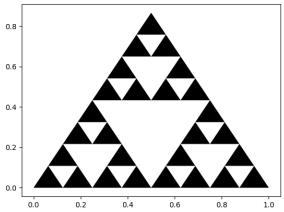


Figure pour $n = 3$

# ALGORITHMES DE TRI

Le programme de CPGE est clair :
- Algorithmes quadratiques : tri par insertion, sélection. Tri par partition-fusion. Tri rapide. Tri par comptage.
- *On fait observer différentes caractéristiques (stable ou non, en place ou non, comparatif ou non, etc).*

►**Selon le programme officiel, aucune connaissance n'est exigible sur les algorithmes de tri.**
Il n'est pas nécessaire d'apprendre le contenu de ce paragraphe par cœur. Néanmoins comprendre le fonctionnement des tris permet de contrôler que qu'on a bien acquis les différents principes de programmation.

► Les tris ont chacun leur fonctionnement propre que l'on peut visualiser avec la vidéo YouTube `https://www.youtube.com/watch?v=kPRA0W1kECg` réalisée par Timo Bingmann.

► On s'intéresse surtout à comparer leur complexité temporelle (temps maximal asymptotique d'exécution) mais aussi leur complexité spatiale (place mémoire maximale prise).

Les algorithmes de tri les moins performants ont une complexité quadratique (en $O(n^2)$) et les plus performants en $O(n\ln(n))$ où $n$ est la taille de la liste donnée en entrée.

► On dit qu'un tri est **comparatif** s'il n'utilise que des comparaisons binaires pour trier la liste. Un bon nombre de tris sont comparatifs.

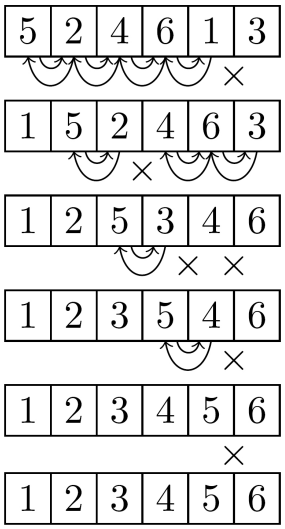
► On dit qu'un tri est **en place** s'il n'utilise qu'un nombre très limité de variables et qu'il modifie directement la structure qu'il est en train de trier. Ce caractère peut être très important si on ne dispose pas de beaucoup de mémoire.

► On dit qu'un tri est dit **stable** s'il préserve l'ordonnancement initial des éléments que l'ordre considère comme égaux.

## TABLEAU COMPARATIF DES TRIS

| Nom du tri | Complexité | En place | Stable | Comparatif |
|---|---|---|---|---|
| Tri bulle | $\mathcal{O}(n^2)$ | oui | oui | oui |
| Tri par sélection | $\mathcal{O}(n^2)$ | oui | non | oui |
| Tri par insertion | $\mathcal{O}(n^2)$ | oui | oui | oui |
| Tri fusion | $\mathcal{O}(n\ln(n))$ | oui | non | oui |
| Tri rapide | $\mathcal{O}(n\ln(n))$ | oui | non | oui |

## TRI BULLE (*Tri utilisant des doubles boucles imbriquées*)




Le **tri bulle** est une stratégie de tri locale, qui consiste à trier une liste d'objets en portant des œillères. Le principe est de regarder les éléments consécutifs de la liste deux par deux et de les mettre dans l'ordre.

On part de la fin de la liste et l'on regarde les éléments consécutivement. A la fin d'un premier passage sur la liste, l'élément le plus petit se retrouve en tête de liste.

C'est de là que provient le nom : l'algorithme imite la remontée de bulles. Si la liste est de taille $n$, après $n$ passages, les $n$ bulles seront remontées et classées dans l'ordre.

Les flèches représentent les échanges réalisés entre les contenus de cases. Une croix indique que les éléments étaient déjà triés et qu'il n'y a donc pas eu d'échange.

Soit `L` une liste de nombres non triée, on code le tri bulle en deux temps (*un algorithme pour effectuer une étape puis un autre pour la répéter*).

- La fonction `remontee_Bulle(L)` déplace l'élément le plus petit en première position en suivant le principe du tri bulle.

```
def remontee_Bulle(L):
    n=len(L)
    for i in range(n-2,-1,-1):
        if L[i+1]<L[i]:
            L[i],L[i+1]=L[i+1],L[i]
    return L
```

- La fonction `Tri_Bulle(L)` permet de trier la liste par appels successifs à `remontee_Bulle(L[i:n])` ou bien en une seule fonction directement (*pas de tranches de listes à faire*) :

```
def Tri_Bulle(L):
    n=len(L)
    for i in range(n):
        L=L[:i]+remontee_Bulle(L[i:n])
    return(L)
```

```
def Tri_Bulle(L):
    n = len(L)
    for i in range(n):
        for j in range(n-2,i-1,-1):
            if L[j+1]<L[j] :
                L[j],L[j+1] = L[j+1],L[j]
    return(L)
```

- La **complexité** de `Tri_Bulle` est **quadratique** : on fait appel $n$ fois à un algorithme de complexité linéaire pour une liste de taille $n-i$, ce qui équivaut à une double boucle triangulaire.
  C'est un des tris les plus lents, il est **en place** et **stable**.

## TRI PAR SÉLECTION (*Tri utilisant des doubles boucles imbriquées*)

- Le **tri par sélection** est une stratégie de tri, qui consiste à trier une liste d'objets en sélectionnant l'élément le plus petit de la liste et en le plaçant en première position, puis en itérant ce procédé avec la partie restante de la liste qui n'a pas été trié.

- A la fin d'un premier passage sur la liste, l'élément le plus petit se retrouve en tête de liste, et à l'étape `i` les $i+1$ premiers éléments de la liste sont triés. À la fin la liste est triée.

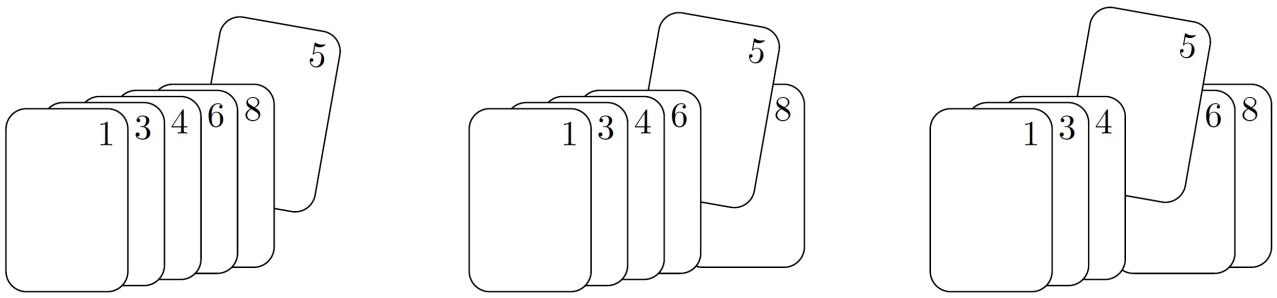
```
def tri_par_selection(L:list)->list:
  """Trie les éléments de L par ordre
  croissant"""
  assert type(L)==list
  for i in range(len(L)):
    ind_min=i
    for j in range(i+1,len(L)):
      if L[j]<L[ind_min]:
        ind_min=j
    L[i],L[ind_min]=L[ind_min],L[i]
  return L
```

**TRI PAR INSERTION** (*Tri utilisant des double boucles imbriquées*)

Le **tri par insertion** est un algorithme efficace de tri d'un petit nombre d'éléments.

Son principe de fonctionnement est celui qui est souvent utilisé pour trier un jeu de cartes. On commence avec la main gauche vide et le paquet de cartes retourné sur la table. On retourne ensuite la première carte du paquet avec la main droite, on l'ajoute à la fin du paquet et on l'amène progressivement à la bonne position dans la main gauche.

Pour trouver cette bonne position, on compare la carte piochée aux cartes de la main gauche une par une, de droite à gauche.




**Ici la main gauche dans laquelle on insère la nouvelle carte retournée numérotée 5.**

A chaque instant, les cartes dans la main gauche sont triées et la main est composée des cartes qui étaient au-dessus du paquet.

▶ Fonction `insertion(L,x)` qui prend en entrée une liste `L` (triée) et un entier ou flottant `x`, ajoute `x` à la liste et l'amène progressivement à la bonne place selon le principe du tri par insertion.

```
def insertion(L,x):
    L.append(x)
    place=len(L)-1
    while L[place]<L[place-1] and place>0:
        L[place],L[place-1]=L[place-1],L[place]
        place-=1
```

▶ Fonction `Tri_insertion(L)` qui effectue le tri par insertion sur une liste `L`.

On commence par définir une nouvelle liste vide `L_triee` que l'on triera au fur et à mesure.

```
def Tri_insertion(L):
    L_triee=[]
    for x in L:
        insertion(L_triee, x)
    return L_triee
```

▶ La complexité de `insertion` est linéaire.

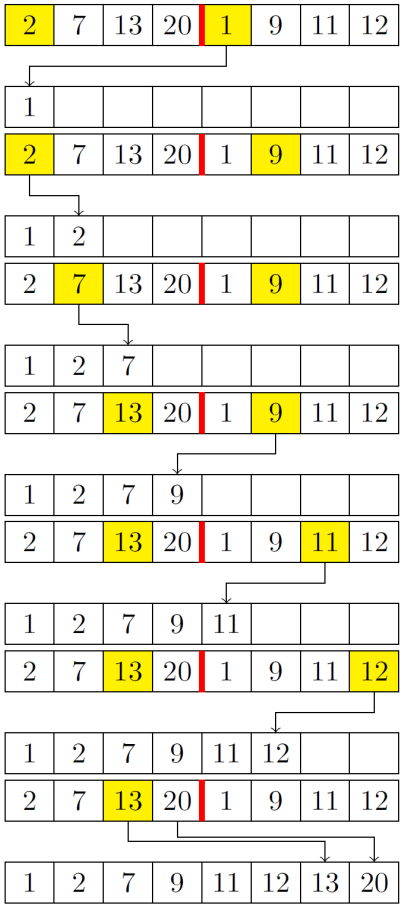
La complexité de `Tri_insertion` est quadratique : on fait appel $n$ fois à un algorithme de complexité linéaire pour une liste de taille $k$ pour $k$ allant de 0 à $n-1$, ce qui équivaut à une double boucle triangulaire en fait.

Il y a donc au maximum $\frac{n(n-1)}{2}$ boucles et une complexité quadratique en $O(n^2)$.

## TRI FUSION (*Tri utilisant la récursivité*)

▶ La fonction `Fusion(L,M)` fusionne deux listes `L` et `M` déjà triées par ordre croissant pour renvoyer la liste globale triée.

```
def Fusion(L,M):
    # On fusionne les deux listes par ordre croissant
    N=[] # On part de la liste vide
    while len(L)!=0 and len(M)!=0:# Si liste non vide
        if L[0]<M[0]: # comparaison entre le 1er de L et M
            N.append(L[0]) # Si c'est L, ajouté à N
            L.pop(0) # Et effacé de L
        else:
            N.append(M[0]) # Si c'est M, ajouté à N
            M.pop(0) # Et effacé de M
    return N+L+M # On renvoie la liste N, complétée.
    # (on ajoute L et M car l'une est vide...)
```

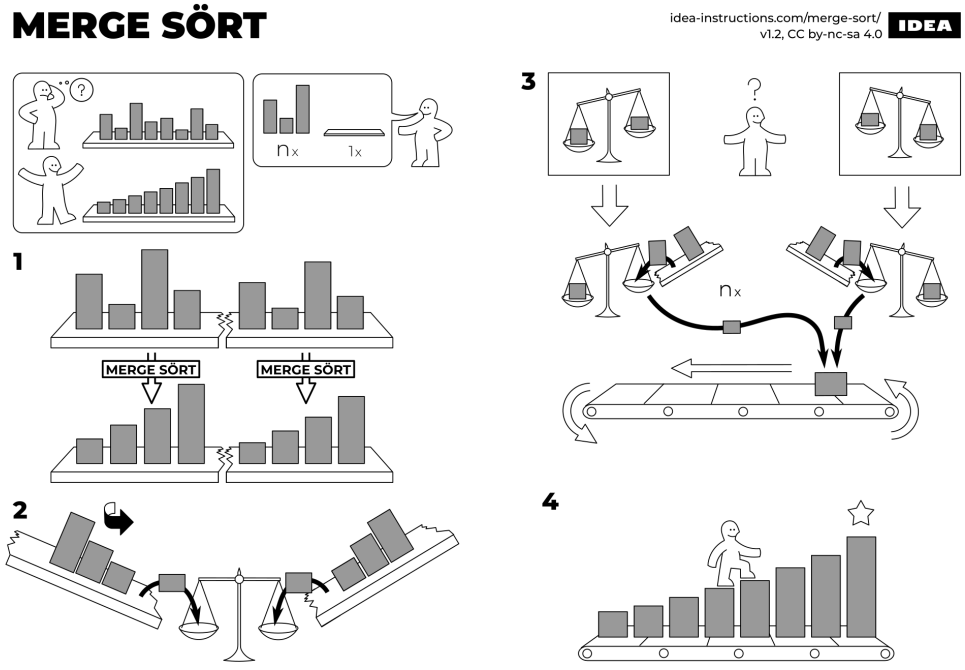



▶ la fonction `TriFusion(L)` qui va trier récursivement la liste `L`.
• Pour cela, si la liste a un élément ou moins, on n'a rien à faire pour la trier.
• Sinon, on coupe la liste au milieu, on trie chacune de ces deux sous-listes de façon récursive.
• Puis on les fusionne les deux sous-listes triées grâce à la fonction Fusion.

```
def TriFusion(L):
  n=len(L)
    if n<=1:
        return L
    return Fusion(TriFusion(L[:n//2]),TriFusion(L[n//2:]))
```

▶ La **complexité** de `Tri_Fusion` est en moyenne en $\mathcal{O}(n\ln(n))$.

▶ C'est un tri efficace, il peut être codé **en place** (*mais pas dans cette version*) mais pas **stable**.



## TRI RAPIDE ou « QUICK SORT » (*Tri utilisant la récursivité*)

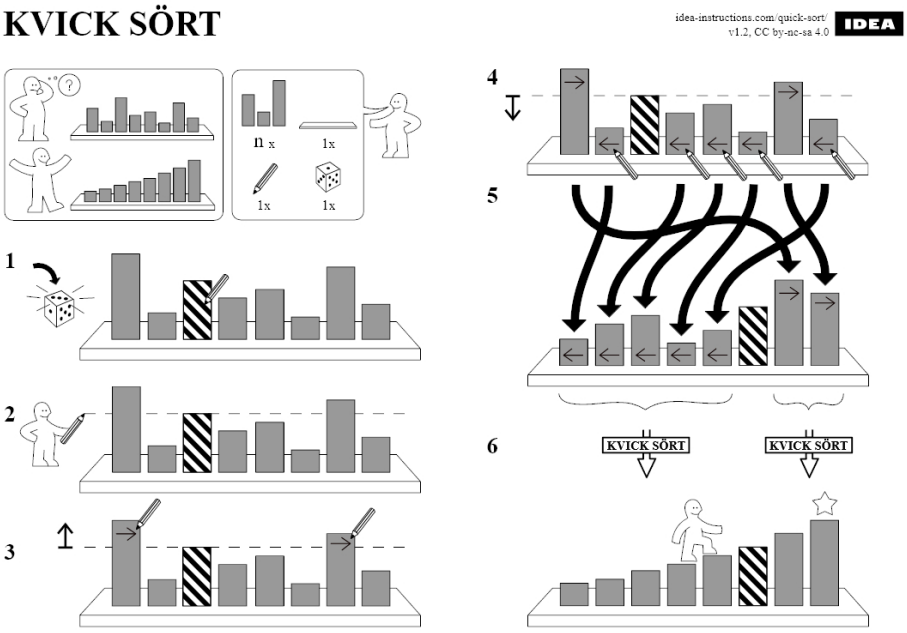
► Le **tri rapide** est une méthode de tri récursive.

*Dans le cadre du tri fusion, la découpe s'opérait en $O(1)$, et la reconstruction en $O(n)$. Ici, la découpe va s'opérer en $O(n)$ et la reconstruction est quasiment instantanée.*

► Le principe est de choisir un **pivot**, autour duquel nous allons couper le tableau.

► Cela consiste donc à mettre tous les éléments plus petits que le pivot à sa gauche, et tout les éléments plus grands que le pivot à sa droite.

► On trie ensuite les sous tableaux gauche et droit : en recollant les morceaux, le tableau est trié.




L'algorithme écrit ici n'est pas en « en place » (vu qu'on utilise des tableaux auxiliaires), néanmoins il peut être écrit de sorte à être **en place**)

► La fonction `Concatene(L,M,N)` renvoie la liste concaténée des éléments de `L`, `M` et `N` dans cet ordre.

1. Une fonction possible est :

```
def concatene(L,M,N):
    return L+M+N
```

► La fonction `Pivot(L)`, où `L` est une liste, qui renvoie trois listes `(G,[L[0]],D)` où `G` est est une liste contenant les éléments de `L` strictement plus petit que `L[0]` et `N` est une liste contenant les autres éléments de `L`.

2. Une fonction possible est :

```
def Pivot(L):
    pivot=L[0]
    G,D=[],[]
    for k in range(1,len(L)):
        if L[k]<pivot:
            G.append(L[k])
        else:
            D.append(L[k])
    return G,[L[0]],D
```

► La fonction `TriRapide(L)` qui va trier `L` de façon récursive, pour cela si `L` n'a pas d'éléments, réfléchir à ce qu'il faut renvoyer. Sinon, diviser la liste `L` grâce à la fonction `Pivot`, trier les listes `G` et `D` de façon récursive, et combiner le tout, grâce à la fonction `Concatene`.

3. Une fonction possible est :

```
def TriRapide(L):
    if len(L)==0:
        return L
    G,P,D=Pivot(L)
    G=TriRapide(G)
    D=TriRapide(D)
    return concatene(G,P,D)
```

► Le tri rapide a été inventé par le scientifique C.A.R. Hoare en 1961.

► **complexité** de `Tri_Fusion` est en moyenne en $\Theta(n\ln(n))$.

► C'est un tri efficace, il est **en place** (si on n'utilise pas de tableaux auxiliaires) mais pas **stable**.

# ALGORITHMES GLOUTONS

Le programme de CPGE est clair :
• Rendu de monnaie. Allocation de salles pour des cours. Sélection d'activité.
• *On peut montrer par des exemples qu'un algorithme glouton ne fournit pas toujours une solution exacte ou optimale.*

Le but des algorithmes gloutons est de résoudre des problèmes d'optimisation à l'aide d'algorithmes. Un problème d'optimisation consiste à minimiser ou maximiser une fonction/quantité sur un ensemble.

On peut citer comme exemple :

- **le problème du sac à dos :** maximiser le contenu (en valeur) d'un sac à dos de volume fixé en le remplissant d'objets ayant chacun un volume et une valeur qui leur est propre.
- **le problème du rendu de monnaie :** comment rendre la monnaie à un client avec le moins de pièces/-billets possible ?
- **le problème d'allocation de salles :** comment caser le maximum d'heures de cours dans le moins de salles possibles tout en restant dans une plage horaire la moins ample possible.
- **optimisation de trajet :** minimiser le temps de trajet entre deux points $A$ et $B$ sur un réseau routier

Tous ces problèmes n'ont pas nécessairement une unique solution et on ne connait pas toujours d'algorithme optimal pour répondre au problème dans toutes les situations de départ. On développe néanmoins une tactique via un certain type d'algorithmes appelés algorithmes gloutons dont le principe est de prendre en priorité les objets « les plus efficaces » pour répondre au problème.

### Exemple. Problème de rendu de monnaie

Soit un système monétaire de pièces/billets : une liste ordonnées par valeur décroissante. Avec l'Euro, on a :
`L=[500,200,100,50,20,10,5,2,1,0.5,0.2,0.1,0.05,0.02,0.01]`
Pour rendre une somme de (94€76 par exemple) en monnaie, on commence par utiliser la plus grande pièce/billet que l'on peut utiliser pour payer ce montant. Puis de réitérer ce procédé jusqu'à avoir totalement rendu la monnaie.

► On écrit une fonction `PlusGrandePiece` qui renvoie la plus grande valeur possible pour rendre le montant.

► On écrit une fonction `GloutonPieces` qui réitère ce procédé jusqu'à avoir rendu tout le montant.

► On rajoute $0,0001$ (*nombre inférieur à la plus petite unité de valeur*) à chaque test sur les flottants pour éviter les problèmes de représentation des nombres dus aux arrondis dans les calculs.

```
def PlusGrandePiece(montant,systeme):
    for l in systeme:
        if l<=montant+0.001:
            return l
```

```
def GloutonPieces(montant,systeme):
    reste=montant
    L=[]
    while reste>0.001:
        piece=PlusGrandePiece(reste,systeme)
        L.append(piece)
        reste=reste-piece
    return L
```

Avec les commandes

```
SystemeEuro=[500,200,100,50,20,10,5,2,1]
SystemeEuro+=[0.5,0.2,0.1,0.05,0.02,0.01]

print(GloutonPieces(94.76,SystemeEuro))
```

on obtient comme résultat

```
[50, 20, 20, 2, 2, 0.5, 0.2, 0.05, 0.01]
```

### Remarque.

Dans un système monétaire `[10,9,2,1]` et pour un montant `18`, cet algorithme n'est pas optimal, en effet, on a été trop gourmand en prenant directement une pièce de 10€ (la plus grande pièce possible), et du coup on ne pourra plus tomber sur la solution optimale (2 pièces de 9€) car il reste 8€ à payer (avece 4 pièces de 2€). Pour cette raison, on dit que c'est un algorithme **glouton**.

# MANIPULATION DE FICHIERS TEXTE

**Selon le programme de CPGE, aucune des commandes ci-dessous n'est à connaitre par cœur.**

Ouvrir un fichier avec Python se fait à l'aide de la commande `open`. Techniquement, on créé un objet Python (*dont le nom est complètement indépendant de celui du fichier texte*) qui est un pointeur vers le fichier.

```
contenu=open("nom_fichier.txt",options à ajouter)
```

La commande `open` est assorti d'options, qui dépend de ce que l'on veut faire avec le fichier une fois ouvert :

- mode lecture (*option* `"r"`) si on souhaite **uniquement lire** le fichier, accéder au contenu sans le modifier ;
- mode écriture (*option* `"w"`) si on souhaite **uniquement écrire** des choses dans le fichier ;
- mode en ajout (*option* `"a"`) si on veut **uniquement ajouter** du contenu à un fichier en contenant déjà.

| Commandes ou syntaxes clés | Description |
|---|---|
| `contenu=open("nom_fichier.txt","r")`<br>`chaine=contenu.read()`<br>`contenu.close()` | Ouvre le fichier texte `nom_fichier.txt` en mode lecture, sous le nom `contenu`. Stocke le contenu dans la variable de type chaîne de caractères `chaine`, puis ferme le fichier. |
| `contenu.read(666)` | Si on souhaite ne pas lire l'intégralité du fichier, on peut préciser un entier dans `read`, et on ne lira qu'un nombre de caractères correspondant à l'argument.<br>Si on fait un nouveau `read` ensuite, la lecture reprendra au caractère où on s'est arrêté au `read` précédent. |
| `contenu.readline()` | permet de lire une ligne du fichier (`\n` en fin de ligne compris). Le curseur de lecture du fichier est maintenant au début de la ligne suivante. |
| `contenu.readlines()` | Créé une liste contenant les différentes lignes du fichier texte (`\n` en fin de ligne compris). |
| `contenu=open("nom_fichier.txt","a")`<br>`contenu.write("blablabla")`<br>`contenu.write("bliblibli")`<br>`contenu.write("blobloblo")`<br>`contenu.close()` | Permet d'ouvrir le fichier `nom_fichier.txt` en mode ajout. écrit les textes `blablabla`, `bliblibli`, `blobloblo` dans le fichier, puis ferme le fichier.<br>*les différents textes seront écrits les uns à la suite des autres à la fin du fichier* |
| `contenu=open("nom_fichier.txt","w")`<br>`contenu.write("Texte␣intéressant")`<br>`contenu.close()` | Permet de créer un fichier accessible à l'écriture.<br>Efface et remplace le fichier de même nom s'il existait déjà. |
| `with open("data.txt","r") as contenu:`<br>`  print(contenu.read())` | Autre syntaxe plus courte qui permet de s'émanciper du problème de fermeture du fichier avec le mot clé `with`. |

**Remarque. ATTENTION !**

• On ne peut pas ouvrir un fichier simultanément en mode lecture et écriture : il faut le refermer avant.

**Remarque. ATTENTION !**

• L'option `"w"` créé automatiquement un nouveau fichier texte dans lequel écrire.
• Mais surtout va écraser un fichier texte déjà existant portant le même nom.
• Si on veut modifier le contenu d'un fichier existant, il faut absolument utiliser l'option `"a"`).

# MATRICE DE PIXELS ET IMAGES

Le programme de CPGE est clair :

- Algorithmes de rotation, de réduction ou d'agrandissement.
- Modification d'une image par convolution : flou, détection de contour, etc.
- *Les images servent de support à la présentation de manipulations de tableaux à deux dimensions.*

**Selon le programme de CPGE, aucune des commandes ci-dessous n'est à connaitre par cœur.**
*En revanche vous devez savoir sous quelle forme est stockée une image dans un fichier.*

## IMAGES MATRICIELLES = MATRICES(LISTES 2D) DE PIXELS

Il existe deux types d'images : les images **vectorielles** et les images **matricielles** :

- **Les images vectorielles** sont (re-)construites à partir d'équations mathématiques et basées sur des formes géométriques. Zoomer ou dézoomer sur une image vectorielle relance le calcul pour affiner la représentation en fonction de la résolution de l'écran. Les formes paraissent lisses (format `.svg`) **Pas au programme de CPGE.**
- **Les images matricielles**, au programme de CPGE, sont des tableaux 2D de points qu'on appelle pixels. Zoomer sur une image matricielle met en évidence la discrétisation des contours.
  Les formats classiques d'images matriciels sont `.bmp`, `.jpg`, `.png`.

**Image `png` en résolution** $390 \times 250$

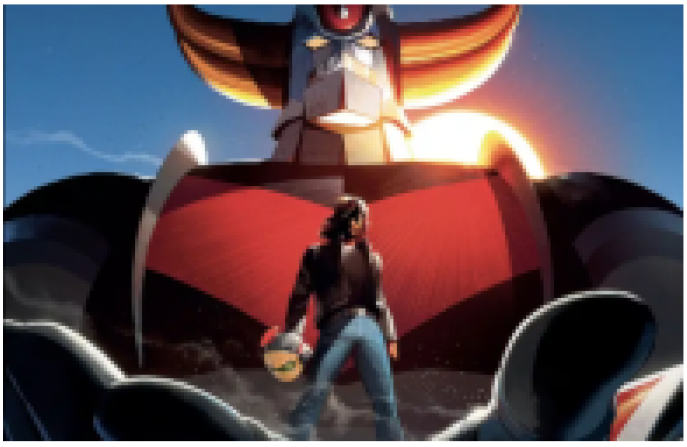


**Image `png` en résolution** $195 \times 125$

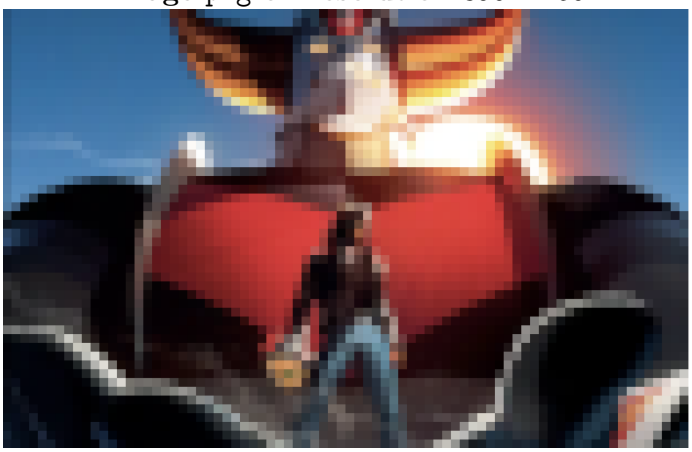


**Image `png` en résolution** $78 \times 50$

**Image `png` en résolution** $39 \times 25$

## EXEMPLE DE COMMANDES D'IMPORT DE FICHIERS IMAGES

```
Img=imageio.imread("noir_et_blanc.png").tolist() #Convertit une image en liste

# Nombre de colonnes (w) et nombre de lignes (h)
print('w=', len(Img[0]),' h=', len(Img))

plt.imshow(Img,cmap='gray',clim=(0,255)) # Création de l'image comme figure
plt.title("Titre à compléter") #Titre de la figure
plt.show() # Ouvre une fenêtre et affiche l'image
```

## IMAGES EN NOIR ET BLANC

▶ La matrice de pixels, ou plutôt tableau 2D des pixels est notée **ligne par ligne** sous la forme :

```
Img=[[178,234,...,97],[56,0,...,111],...,[154,2,...,255]]
```

▶ La couleur de chaque pixel est codée sur **un octet** (c'est-à-dire 8 bits donc $2^8 = 256$ octets possibles), et représente donc une nuance de gris parmi 256 possibles. On appelle cette valeur **la luminosité du pixel** :

- la valeur 0 code la couleur noire,
- la valeur 255 code la couleur blanche,
- les valeurs intermédiaires codent des nuances de gris de plus en plus claires à mesure que la valeur augmente.

▶ Un pixel est donc un entier accessible via la commande `Img[i][j]` ($i$-ème ligne et $j$-ème colonne).

## IMAGES EN COULEUR (*Non exigibles !*)

▶ Pour les images en couleur, on les code également par une matrice de pixels comme celles en niveaux de gris.

▶ **Dans une image couleur, on associe à chaque pixel un triplet de 3 couleurs : rouge, vert, bleu.**

Chacune des 3 couleurs est donnée par un entier entre 0 et 255 (qu'on représente chacun sur un octet).

▶ Ce codage s'appelle RGB pour Red Green Blue et où pour chacune de ces couleurs :

- la valeur 0 indique qu'on met 0% du « pot de cette couleur ».
- la valeur 255 indique qu'on met 100% du « pot de cette couleur ».

**Chaque pixel est donc représenté par un triplet d'octets codant un entier entre 0 et 255.**

Voici quelques exemples pour bien comprendre le codage des couleurs :

| couleur | R | G | B | Octet 1 | Octet 2 | Octet 3 |
|---|---|---|---|---|---|---|
| noir | 0 | 0 | 0 | 0000 0000 | 0000 0000 | 0000 0000 |
| blanc | 255 | 255 | 255 | 1111 1111 | 1111 1111 | 1111 1111 |
| rouge | 255 | 0 | 0 | 1111 1111 | 0000 0000 | 0000 0000 |
| vert | 0 | 255 | 0 | 0000 0000 | 1111 1111 | 0000 0000 |
| bleu | 0 | 0 | 255 | 0000 0000 | 0000 0000 | 1111 1111 |
| violet | 132 | 122 | 191 | 1000 0100 | 0111 1010 | 1011 1111 |

▶ Les images couleur sont donc stockées sous la forme d'un tableau `image` à trois dimensions.

▶ Chaque valeur de couleur associée au pixel correspond au triplet d'octets stocké dans `image[i][j]`.
(pixel en RGB : `[image[x][y][0]` = *Red*, `image[x][y][1]` = *Green*, `image[x][y][2]` = *Blue*])

## QUELQUES EXEMPLES D'ALGORITHMES DE MODIFICATION D'IMAGE

*Ces algorithmes ne sont pas à apprendre par cœur mais sont classiques lors de la manipulation d'image.*

### ▶ Récupérer la résolution de l'image

```
def dim(Img):
    return (len(Img[0]), len(Img))
```

### ▶ Négatif d'une image en niveau de gris

```
def inversion(image):
    w,h=dim(image)
    for y in range(h):
        for x in range(w):
            image[y][x]=255-image[y][x]
    return image
```

### ▶ Appliquer un effet de seuil

```
def noir_et_blanc(image,s):
    w,h=dim(image)
    for y in range(h):
        for x in range(w):
            p = image[y][x]
            if p<=s:
                image[y][x]=0
            else:
                image[y][x]=255
    return image
```

### ▶ Assombrir une image en niveaux de gris

```
def assombrir(image):
    w,h=dim(image)
    for x in range(w):
        for y in range(h):
            image[y][x]=image[y][x]//2
    return image
```

### ▶ Éclaircir une image en niveaux de gris

```
def eclaircir(image):
    w,h=dim(image)
    for x in range(w):
        for y in range(h):
            if image[y][x]*2<256:
                image[y][x]=image[y][x]*2
            else:
                image[y][x]=255
    return image
```

### ▶ Enlever le rouge sur une image couleur

```
def no_rouge(image):
    w,h=dim(image)
    for x in range(w):
        for y in range(h):
            R,G,B=image[y][x]
            image[y][x]=[0,G,B]
    return image
```

### ▶ Effet miroir horizontal sur une image

```
def flip(M):
    w,h = len(M[0]),len(M)
    for i in range(h):
        for j in range(w//2):
            M[i][j],M[i][w-1-j]=M[i][w-1-j],M[i][j]
    return image
```

### ▶ Faire tourner une image de 90°

```
def rotate(image):
    w,h = len(image[0]),len(image)
    image2=[[0 for j in range(h)] for i in
    range(w)]
    for i in range(h):
        for j in range(w):
            image2[j][i]=image[i][w-1-j]
    image=image2
    return image
```

### ▶ Détecter les concours sur une image N&B

```
def energie(M):
    w,h = len(M[0]),len(M)
    e = [[0 for j in range(w)] for i in
    range(h)]
    for i in range(1,h-1):
        for j in range(1,w-1):
            vx = (M[i+1][j] - M[i-1][j])//2
            vy = (M[i][j+1] - M[i][j-1])//2
            e[i][j] = (abs(vx)+abs(vy))
    return e
```

# REPRÉSENTATION DES NOMBRES

Le programme de CPGE est clair :

• Représentation des entiers positifs sur des mots de taille fixe. *La conversion d'une base à une autre n'est pas un objectif de formation.*

• Représentation des entiers signés sur des mots de taille fixe. *Complément à deux.*

• Entiers multi-précision de Python. *On les distingue des entiers de taille fixe sans détailler leur implémentation. On signale la difficulté à évaluer la complexité des opérations arithmétiques sur ces entiers.*

• Distinction entre nombres réels, décimaux et flottants. *On montre sur des exemples l'impossibilité de représenter certains nombres réels ou décimaux dans un mot machine.*

• Représentation des flottants sur des mots de taille fixe. Notion de mantisse, d'exposant. *On signale la représentation de* 0 *mais on n'évoque pas les nombres dénormalisés, les infinis ni les NaN. Aucune connaissance liée à la norme IEEE-754 n'est au programme.*

• Précision des calculs en flottants. *On insiste sur les limites de précision dans le calcul avec des flottants, en particulier pour les comparaisons. Le comparatif des différents modes d'arrondi n'est pas au programme.*

## ENTIERS NATURELS

### ▶ Décomposition en base $b$

L'écriture en base 10 est celle utilisée usuellement : 1981 représente le nombre $1\times10^3+9\times10^2+8\times10^1+1\times10^0$. Pour un entier $b>2$, on peut définir l'unique représentation en base $b$, il existe $0\leqslant a_k\leqslant b-1$ ($k\in[\![0,p]\!]$) :

l'écriture $(a_pa_{p-1}\dots a_0)_b$ représente le nombre : $a_pb^p+a_{p-1}b^{p-1}+\dots+a_1b+a_0$.

Ainsi, en base 3 par exemple, seuls les chiffres $0,1$ et 2 seront utilisés, et :

le nombre $(210122)_3$ représente l'entier $2\times3^5+3^4+3^2+2\times3^1+2$, c'est-à-dire 584.

**Exemple.** Voici par exemple l'écriture de 17 dans toutes les bases entre 2 et 9 :

$$\begin{array}{rcll} 17 &=& 1\times2^4+0\times2^3+0\times2^2+0\times2^1+1\times2^0 &= (10001)_2\\ &=& 1\times3^2+2\times3^1+2\times3^0 &= (122)_3\\ &=& 1\times4^2+0\times4^1+1\times4^0 &= (101)_4\\ &=& 3\times5^1+2\times5^0 &= (32)_5\\ &=& 2\times6^1+5\times6^0 &= (25)_6\\ &=& 2\times7^1+3\times7^0 &= (23)_7\\ &=& 2\times8^1+1\times8^0 &= (21)_8\\ &=& 1\times9^1+8\times9^0 &= (18)_9 \end{array}$$

### ▶ Conversion entre bases

Pour convertir le nombre $x=(a_p\dots a_0)_p$ en base 10 il suffit d'appliquer la formule : $x=\displaystyle\sum_{k=0}^{p}a_kb^k$.

**Exemple :** $(210122)_3=2\times3^5+1\times3^4+1\times3^2+2\times3^1+2\times3^0=584$.

**Remarque :** Sous Python, pour convertir un nombre `x` (écrit sous forme d'une chaîne de caractères) d'une base `b` quelconque en base 10, on utilise `int(x,b)`.

**Algorithme : Écriture d'une base** 10 **en base** $b$
**Entrée :** Un entier $N > 0$ et en entier $b \geqslant 2$.
**Sortie :** L'écriture de $N$ dans la base $b$.
$i \leftarrow 0$
$M \leftarrow N$
**tant que** $M \neq 0$ **faire**
    $(a, r) \leftarrow$ quotient et reste de la division euclidienne de $M$ par $b$
    $a_i \leftarrow r_i$
    $M \leftarrow q$
    $i \leftarrow i + 1$
**retourner** $(a_p, a_{p-1}, \ldots, a_0)$ (*si le dernier indice de $i$ est $p$*)

Écrivons 584 en base 3. On a :

$584 = 3 \times 194 + 2$,

$194 = 3 \times 64 + 2$,

$64 = 3 \times 21 + 1$,

$21 = 3 \times 7 + 0$,

$7 = 3 \times 2 + 1$ et

$2 = 0 \times 3 + 2$.

Ainsi $584 = (210122)_3$.

### ▶ Décomposition en base 2

En base 2, seuls les chiffres 0 et 1 sont donc utilisés, ce qui est adapté pour le codage informatique. Les opérations se généralisent alors en base 2 :

$$\begin{array}{cccccccccc} & & & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ + & & & & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

$$\begin{array}{cccccc} & & & 1 & 0 & 1 & 0 \\ \times & & & & 1 & 0 & 1 \\ \hline & & & 1 & 0 & 1 & 0 \\ & 1 & 0 & 1 & 0 & . & . \\ \hline & 1 & 1 & 0 & 0 & 1 & 0 \end{array}$$

### ▶ Décomposition en base 16

Pour représenter les « chiffres » manquants (de 11 à 15), on utilise les lettres de $a$ à $f$ :

| lettre | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |
|---|---|---|---|---|---|---|
| signification | 10 | 11 | 12 | 13 | 14 | 15 |

**Remarque :** Utilité ? l'écriture binaire est très longue à écrire et la conversion base 2 à base 16 très rapide :

| hexadécimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| binaire | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |

| hexadécimal | 8 | 9 | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |
|---|---|---|---|---|---|---|---|---|
| binaire | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

Aussi, pour convertir un nombre quelconque de la base 2 à la base 16, il suffit de regrouper les chiffres qui le composent par paquet de 4 et convertir chacun de ces paquets en un chiffre en base 16 (1 octet = 2 caractères hexadécimaux).

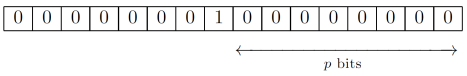
**Exemple :** $(1011\ 0110\ 1110\ 1001)_2 = (b6e9)_{16}$.
En code RGB un navigateur web va interpréter le code couleur $(ffa500)_{16}$ comme du orange, $((00ff00)_{16} =$ vert, $(ee82ee)_{16} =$ violet).

### ▶ Codage d'un entier naturel sur $n$ bits

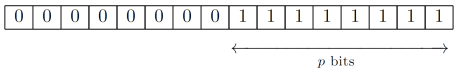
Le codage dans l'ordinateur doit s'effectuer sur un nombre de bits fixé $n$ (différent des mathématiques). Le principal problème est la limitation de la taille du codage.

Un codage sur $n$ bits permet de représenter tous les nombres naturels compris entre 0 et $2^n - 1$.

**Exemple :**
• Ainsi, un octet permet de coder les entiers allant de $0 = (00000000)_2$ à $255 = (11111111)_2$.
• En 64 bits (soit 8 octets) tous les nombres de $0 = (0000000000000000)_{16}$ à $2^{64} - 1 = (ffffffffffffffff)_{16}$.
• L'entier naturel $2^p$ est représenté par :




• L'entier naturel $2^p - 1$ est représenté par :



**▶ Dépassement de capacité**

On représente sur $n$ bits les entiers naturels dans $[\![0, 2^n-1]\!]$. Par opération sur ces entiers, il est possible que le résultat du calcul n'appartienne plus à cet intervalle !

**Exemple.** Sur une addition dépasse la borne supérieure en considérant la somme :

$$\begin{array}{cccccccccc} & & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ + & & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ \hline & \boxed{1} & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Il faudrait donc 9 bits pour représenter la somme. ($149 + 142 = 291 > 255$, le 1 encadré est tronqué et on obtiendrait $49 + 142 = 291 - 2^8 = 35$ !
Ce problème de **dépassement de capacité** est la raison du crash d'Ariane 5 en 1996...

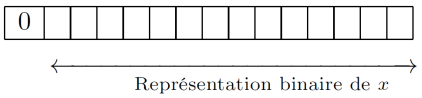
## ENTIERS RELATIFS

Il est nécessaire de coder le signe de ce dernier sur un bit (0 pour les nombres + et 1 pour les nombres −)

Le processeur va réserver le premier bit pour le signe, et les $n-1$ autres bits pour l'entier à proprement parler.

Le plus grand entier relatif positif représentable sur $n$ bits est donc égal à $2^{n-1}-1$.

En base 2, il s'écrit $(0\ \underbrace{111\ldots11111}_{n-1 \text{ chiffres}})_2$.



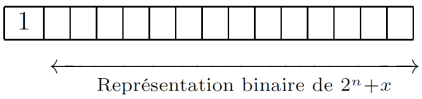

Un entier relatif positif $x$ est alors représenté par :

**▶ Codage des entiers négatifs : complément à 2**

**Attention :** Il ne suffit pas d'ajouter un bit de signe devant le code de l'entier naturel $|n|$ ! (0 *aurait deux représentations et l'addition ne s'appliquerait pas !*)

C'est pourquoi on utilise le codage particulier pour représenter les entiers négatifs, dit en **complément à deux** :

le nombre négatif $x$ est représenté en mémoire par la représentation binaire de l'entier (positif) $2^n + x$.




Un entier relatif négatif $x$ est alors représenté par :

**Exemple.** Pour simplifier les calculs, considérons une représentation sur 8 bits ($n = 8$).
L'entier relatif 105 est représenté par l'octet 01101001, ce qui correspond à la représentation en base 2 de l'entier $105 = 2^6 + 2^5 + 2^3 + 2^0$. L'entier relatif $-105$ est représenté par l'octet 10010111, ce qui correspond à la représentation en base 2 de l'entier $256 - 105 = 151 = 2^7 + 2^4 + 2^2 + 2^1 + 2^0$.

Pour que cette représentation sur $n$ bits commence par un 1, il est donc nécessaire que $2^n + x$ soit compris entre $2^{n-1} = (1\ \underbrace{000\ldots00000}_{n-1 \text{ chiffres}})_2$ et $2^n - 1 = (1\ \underbrace{111\ldots11111}_{n-1 \text{ chiffres}})_2$.

Car $2^{n-1} \leqslant 2^n + x \leqslant 2^n - 1$ si, et seulement si, $-2^{n-1} - 1 \leqslant x \leqslant -1$.
Le plus petit entier négatif représentable sur une architecture à $n$ bits est donc égal à $-2^{n-1}$ est représenté par :

$$2^{n-1} = (1\ \underbrace{000\ldots00000}_{n-1 \text{ chiffres}})_2$$

Ainsi, les entiers relatifs représentables sur une architecture à $n$ bits sont compris entre $-2^{n-1}$ et $2^{n-1}-1$.

**▶ Opérations sur les entiers signés**

**• L'addition**

Grâce à la représentation par complément à deux, tout entier dans $[\![-2^{n-1}, 2^{n-1}-1]\!]$ à une unique représentation. Autre intérêt majeur : **additionner deux nombres relatifs revient à additionner leurs représentations en ne gardant que les $n$ bits de poids faible**.

**Exemple :** Somme de $-91$ et de 113 avec un codage sur 8 bits (on a $2^8 = 256$.
$-91$ est négatif donc il est représenté par son complément à deux $256 - 91 = 165 = (10100101)_2$.
113 est positif donc il est représenté par sa décomposition en base 2 : $113 = (01110001)_2$.
On additionne ces deux représentations $-91 + 113 = (\boxed{1}00010110)_2$ et on ne garde que les 8 derniers bits, à savoir $(00010110)_2$.

**• L'opposé (et donc la soustraction car a-b = a + (-b) ...)**

Pour obtenir la représentation de l'opposé de $x$, il suffit de :

— **avoir la représentation de $x$ et remplacer tous les bits 0 en des bits 1 et réciproquement.**

— **additionner 1 au résultat obtenu.**

**Exemple :** $x = 113$ représenté par 01110001, donc $-x$ est représenté par $10001110 + 1 = 10001111$.

**▶ Réponse au dépassement de capacité : les entiers multi-précision de Python**

Dans un ordinateur, on utilise des registres de

— 32 ou 64 bits,

— soit des entiers relatifs dans $[\![-2^{31}, 2^{31}-1]\!]$ ou $[\![-2^{63}, 2^{63}-1]\!]$.

Si le résultat d'un calcul ne rentre pas dans l'intervalle, le résultat est-il erroné ? La réponse varie !

• Dans un langage de bas niveau comme le C, le type `int` code des entiers de 32 bits et il peut y avoir des dépassements de capacité ($3^{20} = -808182895$ qui est égal à $3^{20} - 2^{32}$ par exemple)

• sur votre calculatrice, dépendant de votre modèle, le nombre de bits maximal est différent, et bien qu'assez élevé, est limité et il peut y avoir des dépassements de capacité.

• En revanche l'interprète Python détecte le débordement de capacité : lorsque le résultat d'un calcul dépasse le plus grand entier représentable en complément à deux (à savoir $2^{63} - 1$), la représentation des entiers change.

On adopte alors la représentation sous forme dite des **entiers longs** ou **multi-précision**.

Les longs entiers sont en fait codés par paquets de bits de longueur fixée, et il peut y avoir un nombre potentiellement infini de paquets (limité par la mémoire, naturellement). Tout ceci se fait de manière transparente pour l'utilisateur, on n'aura donc pas à se soucier des dépassements de capacité lorsqu'on manipulera des entiers.

Contrairement au complément à deux, cette représentation n'est pas limitée en taille. Cela présente bien évidemment des avantages, mais aussi des inconvénients : les opérations sur les entiers longs prennent plus de temps que sur les entiers classiques.

## NOMBRES RÉELS

### ▶ Décimaux

• Les nombres décimaux sont représentables dans n'importe quelle base $b \geqslant 2$ par une décomposition unique.

**Exemple.**
Par exemple, en base 10, on a $\frac{1}{3} = 0,3333333333\ldots$ et $\frac{49}{3} = 16 + \frac{1}{3} = 16,3333333333$.
En base 2, on a $\frac{1}{3} = (0,0101\ 0101\ 0101\ 0101\ 0101\ldots)_2$ et $\frac{49}{3} = 2^4 + \frac{1}{3} = (1000,0101\ 0101\ 0101\ 0101\ 0101\ldots)_2$.

**Exemple.** $x = 0,1$ est un nombre décimal mais n'a pas une représentation binaire finie ! On peut vérifier que :

$$(0,0\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ldots)_2 = 0,1$$

• **Conversion de la base $p$ à la base 10**

Si $x = (x_p x_{p-1}\ldots x_0, b_1 b_2 b_3\ldots)$, alors pour passer de la base $b$ à la base 10 il suffit d'appliquer la formule :

$$x = \sum_{k=0}^{p} x_k \times b^k + \sum_{k=1}^{+\infty} \frac{a_k}{b^k}$$

**Exemple.** Si $x = (0,00011)_2$ alors en base 10, $x = 1 \times 2^{-4} + 1 \times 2^{-5} = \frac{1}{16} + \frac{1}{32} = \frac{3}{32} = 0,9375$.

• **Conversion de la base 10 à la base $p$**

**Exemple.** Soit $x = 78,347$ de partie entière 78 et de partie fractionnaire $0,347$.
On a déjà vu comment convertir 78 en binaire (divisions euclidiennes par 2 successives).
On trouve après calculs $78 = (100\ 1110)_2$.
Pour la partie fractionnaire on va faire à l'inverse des multiplications successives par 2 :
$0,347 \times 2 = 0,694 < 1$ je pose 0 et $0,347 = 0,0\ldots$
$0,694 \times 2 = 1,388 > 1$ je pose 1 et $0,347 = 0,01\ldots$
$0,388 \times 2 = 0,766 < 1$ je pose 0 et $0,347 = 0,010\ldots$
$0,776 \times 2 = 1,552 > 1$ je pose 1 et $0,347 = 0,0101\ldots$
$0,552 \times 2 = 1,104 > 1$ je pose 1 et $0,347 = 0,01011\ldots$
$0,104 \times 2 = 0,208 < 1$ je pose 0 et $0,347 = 0,010110\ldots$
$0,208 \times 2 = 0,416 < 1$ je pose 0 et $0,347 = 0,0101100\ldots$
$0,416 \times 2 = 0,832 < 1$ je pose 0 et $0,347 = 0,01011000\ldots$
On continue ainsi jusqu'à trouver la "période" de son développement en binaire ou bien tomber sur 1 exactement.

### ▶ Réels

Comment exprimer des nombres réels en machine ? La représentation scientifique utilisée ci-dessus s'y prête bien : on garde un certain nombre de chiffres significatifs, et on peut représenter des nombres très petits (en valeur absolue) ou très grands en jouant sur l'exposant dans la puissance de 10, qui peut être négatif ou positif.

**Exemple.** Prenons quelques constantes physiques célèbres :

— La vitesse de la lumière dans le vide : $c_0 = 2,99792458 \times 10^8 m.s^{-1}$.

— Constante gravitationnelle : $G = 6,67384 \times 10^{11} m^3.kg^{-1}.s^{-2}$.

— Nombre d'Avogadro : $N_A = 6,02214129 \times 10^{23} mol^{-1}$.

— Constante des gaz parfaits : $R_0 = 8,3144621 J.K^{-1}.mol^{-1}$.

L'important est donc de pouvoir représenter des réels d'ordres de grandeur très différents, en gardant une précision suffisante pour chacun et la représentation scientifique permet cela.

### ▶ Flottants

• L'écriture en binaire d'un nombre décimal peut être infini ($0,1$ par exemple). Ainsi en machine, sa représentation machine sera nécessairement tronquée et ne correspondra pas exactement au nombre (mais en sera néanmoins très proche).

• La conversion d'un nombre décimal en sa représentation machine va donc souvent provoquer une approximation qui dans certains cas conduit à des résultats qui peuvent paraître étranges à un utilisateur non averti.

**Exemple.** Observons ce calcul en Python :

```
>>> 0.1+0.2
0.30000000000000004
>>> (0.1+0.2)-0.3
5.551115123125783e-17
```

Ainsi un calcul sur des nombres décimaux sera toujours entaché d'une certaine marge d'erreur dont il faudra tenir compte !

```
>>> 0.1+0.2==0.3
False
```

Une conséquence importante : l'égalité entre nombres flottants n'a aucun sens : Quand on utilise le type `float`, il est rarissime d'utiliser l'opérateur de test `==`. On fera toujours intervenir une marge d'erreur (absolue ou relative).

```
>>> abs(0.1+0.2-0.3)<1e-10
True
```

Ici, on a choisi une marge d'erreur absolue en considérant que toute quantité inférieure à $10^{-10}$ est nulle.

### ▶ Mantisse, Exposant, exemple de la norme IEEE-754

**Rien n'est à retenir par cœur concernant cette norme mais comprenez le principe**

La norme IEEE-754 actuellement le standard pour la représentation des nombres à virgule flottante en binaire. Nous allons en donner une description pour une architecture 32 et 64 bits. (64 bits est la plus courante.)
Même principe que la notation scientifique en base 10 d'un nombre décimal :

— en base 10 : signe, multiplié par un nombre décimal de l'intervalle $[1,10[$, multiplié par une puissance de 10,

— en base 2 : un nombre décimal non nul s'écrira comme un signe, multiplié par un nombre à virgule de l'intervalle $[1,2[$, multiplié par une puissance de 2.

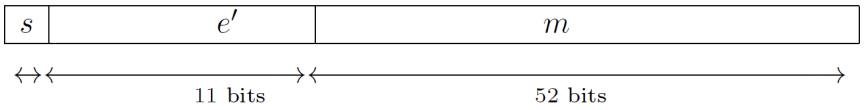
C'est sous cette forme que sont représentés les nombres réels (leur approximation décimale) en machine.

Dans cette norme, les nombres dyadiques sont codés **sur 64 bits** en réservant :

— 1 bit pour le signe,

— 11 bits pour l'exposant.

— 52 bits pour la mantisse.




**Exemple.** $6,25 = (110,01)_2$ a pour représentation normalisée $+(1,1001)_2 \times 2^2$.
Et $0,375 = (0,011)_2$ a pour représentation normalisée $+(1,1)_2 \times 2^{2-}$.

L'exposant est un entier relatif, mais pour permettre une comparaison plus aisée des nombres flottants, il n'est **pas codé suivant la technique de complément à deux mais suivant la technique du décalage** : l'exposant $e$ est représenté en machine par l'entier positif $e' = e + 2^{10} - 1$. Un entier naturel codé sur 11 bits est compris entre 0 et $2^{11} - 1$ donc a priori :

$$0 \leqslant e' \leqslant 2^{11} - 1 \Leftrightarrow -1023 \leqslant e \leqslant 1024$$

(*Les valeurs extrêmes* $e' = 0$ *et* $e' = 2^{11} - 1$ *sont réservées à la représentation de certaines valeurs particulières.*)

**Remarque.** On donne dans le tableau suivant le nombres de bits dévolus à la mantisse et à l'exposant décalé $e'$ dans les représentations sur 32 et 64 bits :

| format | signe | taille de $e'$ | décalage $d$ | taille de la mantisse $m$ | signification |
|---|---|---|---|---|---|
| 32 bits | 1 bit | 8 bits | $2^{8-1} - 1 = 127$ | 23 bits | $(-1)^s \times \underbrace{(1, m_1 \ldots, m_{23})_2}_{\text{mantisse}} \times 2^{e'-127}$ |
| 64 bits | 1 bit | 11 bits | $2^{11-1} - 1 = 1023$ | 52 bits | $(-1)^s \times \underbrace{(1, m_1 \ldots, m_{52})_2}_{\text{mantisse}} \times 2^{e'-1023}$ |

**Exemple.** **Passer de l'écriture décimale à la norme IEEE-754 en 32 bits.**
Donnons la représentation du nombre $21,625$ sur 32 bits. Ce nombre est un dyadique, qui s'écrit :

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

Autrement dit, $21,625 = (10101,101)_2 = (1,0101101)_2 \times 2^4$.

- Le signe est positif, le bit correspondant est donc 0.
- Les 23 bits de la mantisse sont obtenus en complétant 0101101 avec des zéros à droite. (*et oui, pas à gauche comme pour les entiers puisque l'on rajoute des* 0 *après la partie fractionnaire.*)
- L'exposant décalé $e'$ est obtenu en rajoutant à 4 le décalage 127 et en convertissant ce nombre en binaire. Ainsi $e' = 131 = (10000011)_2$.

Par suite, $21,625$ est représenté sur 32 bits comme 0|100 0001 1|010 1101 0000 0000 0000 0000.

**Exemple.** **Passer de la norme IEEE-754 à l'écriture décimale en 32 bits.**
Inversement, considérons le nombre représenté par 1|001 0011 1|001 0000 0000 0000 0000 0000.

- Son bit de signe est 1 : c'est un nombre négatif.
- Son exposant décalé est 00100111 soit 39. En retirant le décalage, on a donc $e - d = -88$.
- Sa mantisse est représentée par 0010… soit $(1,001)_2 = 1 + 2^{-3} = 1,125$.

On obtient donc le nombre $-1,125 \times 2^{88}$, soit environ $-3,6350710512584224 \times 10^{-27}$.

**▶ Erreurs liés à la représentation : Changement de base, absorption**

Le fait que les réels ne soient représentés qu'approximativement fait que les égalités mathématiques ne tiennent plus avec des flottants. Voici trois exemples de ce qu'on peut obtenir en Python (sur 64 bits) :

```
>>> a=0.1
>>> b=0
>>> for i in range(10):
...       b=b+a
>>> b==1
False
```

```
>>> a=2.**100
>>>a==a+1
True
```

```
>>> a,b,c=1,2**-53,1
>>> a+b-c==a-c+b
False
```

- **Changement de base**

Pour le premier exemple, $0,1$ n'est représenté en mémoire que sous forme arrondie. La boucle à pour objet de calculer $10 \times 0,1$ en faisant 10 additions. Les erreurs d'approximation se cumulent, et au final on obtient un résultat très proche de 1, mais qui n'est pas 1 (j'obtiens $1 - 2^{-53}$).

- **Absorption**

Pour le deuxième, l'explication est la suivante : sur 64 bits, il y a 52 bits de mantisse. Le plus petit flottant strictement supérieur à $2^{1000}$ est donc $(1+2^{-52})\times 2^{1000}$. Ainsi, $2^{1000}+1$ est indiscernable de $2^{1000}$. Plus exactement le résultat de l'addition $2^{1000}+1$ est arrondi au flottant le plus proche, à savoir $2^{1000}$ lui même.

D'où l'égalité `a==a+1` qui peut paraître choquante, puisqu'elle démontrerait au passage que $0 = 1$ !

- **Test d'égalité entre flottants**

Pour le dernier exemple, les opérations $+$ et $-$ ayant même priorité, elles sont évaluées de gauche à droite... Or ici $1 + 2^{-53}$ est arrondi au flottant le plus proche, à savoir 1 lui-même.
Donc, `a+b-c` vaut exactement zéro. Par contre, `a-c+b` vaut `b`, car `a` et `c` sont tous deux égaux (à 1).
Il faut être conscient que dans le monde des flottants, les égalités mathématiques ne sont plus vérifiées « qu'à $\varepsilon$ près », à cause des erreurs d'arrondi et de l'impossibilité de représenter de manière exacte les réels. La leçon à retenir des exemples et la suivante : sauf cas particuliers bien précis,

**Le test d'égalité entre deux flottants n'est, en général, pas pertinent !**

- **Cancellation**

Un autre problème se présente lors de la soustraction de deux quantités très proches. Pour fixer les idées, supposons que l'on connaisse deux quantités voisines `x` et `y` avec une précision de 20 chiffres significatifs après la virgule.

```
x = 1,10010010000111111011
y = 1,10010010000111100110
x-y = 0,00000000000000010101
```

Le résultat `x-y` sera normalisé pour être représenté en machine par $(1,0101)_2 \times 2^{-16}$.
Autrement dit, la précision ne sera plus que de 4 chiffres après la virgule ! (contre 20 pour `x` et `y`.)
Cette perte drastique de précision porte le nom de **cancellation**, car la différence de deux quantités très voisines fait littéralement s'évanouir les chiffres significatifs.
Par exemple, il est préférable de calculer $\frac{1}{x(x+1)}$ au lieu de $\frac{1}{x} - \frac{1}{x+1}$ ! En règle générale :

1. On évite d'additionner deux quantités dont l'écart relatif est très grand.
2. On évite de soustraire deux quantités très voisines.

# MÉTHODE DE PROGRAMMATION

La **signature** est l'ensemble des paramètres (avec leurs noms, positions, valeurs par défaut et annotations), ainsi que l'annotation de retour d'une fonction. (informations décrites à droite du nom de fonction.)

**Exemple.** Voici une même fonction mais dont les types des variables ont été modifiées dans les spécifications.

**Spécification 1 :**

```
def addition(a:int, b:int) -> int:
    return a + b
```

**Spécification 2 :**

```
def addition(a:list, b:list) -> list:
    return a + b
```

La signature des deux fonctions ci-dessus sont `(a:int, b:int) -> int` et `(a:list, b:list) -> list`. Cela ne gêne en rien l'exécution de la fonction si les variables entrées ne sont pas du bon type.

Les **annotations** sont des commentaires permettant d'expliquer la démarche/logique du code écrit et rappelant le but du script écrit dans le but d'être bien compris par une personne extérieur.
Placé en tête d'une fonction, l'annotation sert à décrire l'usage de la fonction (accessible avec la commande `help`).

**Exemple.** Voici une même fonction mais dont les types des variables ont été précisés dans les annotation.

**Sans annotation :**

```
def addition(a,b):
    return a + b
```

**Avec annotation**

```
def addition(a,b):
    "Return␣the␣sum␣of␣the␣two␣numbers␣‘a‘␣and␣‘b‘"
    return a + b
```

La commande `help(addition)` renverra alors le texte `Return the sum of the two numbers ‘a‘ and ‘b‘.`

La commande d'**annotation** `assert` est l'un des outils qui peuvent permettre de détecter en cours d'exécution d'un code des erreurs qui conduiraient à un mauvais résultat via un test à réaliser.
Cela permet détecter les erreurs et de sortir de la fonction en indiquant quel type d'erreur a été commis.

**Exemple.**

```
def f(L):
  print(L[0])
```

La commande `f([1,2,3])` renvoie `1`
mais la commande `f([])` renvoie
`IndexError: list index out of range`

Lorsque la liste est vide, il y a une erreur. On peut faire en sorte de vérifier si la liste est vide avant d'appeler `L[0]` et de renvoyer un message souhaité si elle est vide, ce qui arrêtera l'exécution du code.

```
assert booleen, "message␣si␣False"
```

On modifie alors la fonction et le message d'erreur renvoyé est celui de l'échec du test de `assert` :

```
def f(L):
  assert type(L)==list,"Pas␣une␣liste"
  assert len(L)>0,"Liste␣vide"
  print(L[0])
```

La commande `f([1,2,3])` renvoie `1`
la commande `f([])` renvoie maintenant
`AssertionError: Liste vide` et la commande `f(3)` renvoie maintenant
`AssertionError: L n'est pas une liste`

# TERMINAISON ET CORRECTION

Le programme de CPGE est clair :
• La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine.
• On montre sur plusieurs exemples que la terminaison peut se démontrer à l'aide d'un variant de boucle.
• Sur plusieurs exemples, on explicite, sans insister sur aucun formalisme, des invariants de boucles en vue de montrer la correction des algorithmes.

## TERMINAISON

▶ Vérifier la **terminaison d'un algorithme** consiste à vérifier qu'il a bien une fin. Autrement dit, il faut que le nombre d'itérations soit fini.

▶ Pour établir la terminaison d'un algorithme on exhibe un **variant de boucle** qui est une quantité positive, à valeurs dans $\mathbb{N}$, dépendant des variables de la boucle, qui décroît strictement à chaque passage dans la boucle.

**Exemple.**

for

```
1 n = 100
2 L = [i for i in range(n)]
3 Somme = 0
4 for i in range(len(L)):
5   Somme += L[i]
6 print(Somme)
```

`len(L)-i` **est un variant de boucle.**
C'est un entier positif défini dans l'intervalle $[\![0, n-1]\!]$ dont la valeur décroit strictement à chaque itération (`i+=1`).
**La terminaison est vérifiée.**

▶ Lorsqu'une boucle `for` est utilisée avec un compteur non perturbé (pas de modification dans la boucle), on connaît à priori le nombre d'itérations qui sont réalisées, on sait donc que l'algorithme se finit nécessairement.

**Exemple.** Mais il peut y avoir des problèmes si l'on touche au compteur !

for

```
1 L = [0]
2 for Terme in L:
3   L.append(Terme)
```

**Danger :** On modifie la taille de `L` dans la boucle !
A l'itération n°$i$, la taille $t_i$ de `L` vaut $t_i = i+1$ au début de la boucle.
On appelle l'élément d'indice `i` tant que $i < t_i$ (toujours vrai)
**La terminaison n'est pas vérifiée.**

• Lorsqu'une boucle `while` est utilisée, l'algorithme se termine si la condition d'entrée n'est plus vérifiée. Par construction, la condition est vérifiée lors de l'entrée dans l'algorithme afin d'en exécuter le contenu. Il est donc nécessaire de faire évoluer la condition dans la boucle et d'être sûr qu'elle finira par prendre la valeur `False`.

while

```
1 # Division euclidienne a = bq+r
2 a, b, q, r = 25, 7, 0, a
3 while r >= b:
4   r , q = r-b , q+1
5 print(q,r)
```

**Le reste `r` est un variant de boucle.**
`r` $\in \mathbb{N}$ (imposé par `r > b`) et même `r` $\in [\![b, +\infty[\![$ dont la valeur décroit strictement à chaque itération (`r = r - b`).
**La terminaison est vérifiée.**

while

```
1 N = 23
2 while N != 1:
3   N += -2
4 print(N)
```

`N` décroit à chaque itération mais...
... la condition `N != 1` définit l'intervalle $\mathbb{N}\backslash\{1\}$.
On ne répond pas à la condition de terminaison.
La boucle a une fin $\Leftrightarrow$ `N` est impair au départ.

## CORRECTION

▶ La **correction d'un algorithme**, quels que soient les entrées vérifiant sa spécification, consiste à vérifier qu'il renvoie la valeur attendue, que **l'action de l'algorithme correspond à ce qui est attendu**. Cela revient à en faire la démonstration.
On parle de **correction partielle** quand le résultat est correct lorsque l'algorithme s'arrête.
On parle de **correction totale** si elle est partielle et que l'algorithme termine.

▶ Pour les blocs conditionnels (`if`, `elif`, ..., `else`), il n'y a pas grand chose à dire de plus que le bloc lui-même.

| | Programme | Correction |
|---|---|---|
| Programme normal | `# Partie positive`<br><br>`X = 10`<br>`Y = (X+abs(X))/2`<br>`print(Y)` | Il suffit de vérifier que le programme fait ce qu'il annonce :<br>$\frac{X+\lvert X\rvert}{2} = \begin{cases} \frac{X+X}{2} = X \text{ si } X > 0 \\ \frac{X-X}{2} = 0 \text{ si } X > 0 \end{cases}$<br>**Ce programme est correct.** |

▶ En revanche, analyser les boucles `for` et `while` est essentiel, car l'action de ces boucles n'est pas évidente !

▶ On s'assure alors de la correction d'un algorithme à l'aide d'un **invariant de boucle** qui est une propriété $\mathcal{P}$, qui doit être vérifiée tout au long du déroulement de l'algorithme (vraie avant la boucle et qui reste vraie à chaque itération).

**Remarque.** Prouver que l'on a bien un invariant de boucle revient à faire une démonstration par récurrence !

| Étape | Boucle `for` | Boucle `while` |
|---|---|---|
| **Initialisation** | $\mathcal{P}$ vraie à la première itération : $\mathcal{P}(0)$ | $\mathcal{P}$ vraie avant la première itération : $\mathcal{P}(0)$ |
| **Transmission Conservation** | Si $\mathcal{P}$ vraie à l'itération $i$, $\mathcal{P}$ vraie à l'itération $i+1$ : $\mathcal{P}(i) \Rightarrow \mathcal{P}(i+1)$. | Si $\mathcal{P}$ vraie avant l'itération $i$, $\mathcal{P}$ vraie après : $\mathcal{P}(i) \Rightarrow \mathcal{P}(i+1)$ |
| **Sortie Conclusion** | A la dernière itération $n$, $\mathcal{P}$ doit permettre de démontrer que le résultat obtenu est le résultat attendu : $\mathcal{P}(n) \Rightarrow$ Résultat | Après la dernière itération $n$ (lorsque la condition devient fausse), $\mathcal{P}(n)$ doit permettre de démontrer que le résultat obtenu est le résultat attendu : $\mathcal{P}(n) \Rightarrow$ Résultat |

**Remarque.** Pour les boucles `for` :
• Naturellement avec `for i in range(n)`, l'indice de récurrence est le compteur de la boucle.
• On s'adaptera si on utilise `for i in range(m,n)`, (`i` commence à `m`) et `for i in range(m,n,p)` (pas `p`).
• Avec la syntaxe `for x in L`, (où `L` liste), il est nécessaire de faire appel à l'indice de `x` dans `L`.

**Remarque.** Pour les boucles `while` :
• $i = 0$ correspond à l'état initial, avant la première itération.
• $i = 1$ correspond donc à la fin de la première itération, contrairement à la correction de la boucle for (où $i = 1$ correspond à la fin de la seconde itération, $i = 0$ étant la fin de la première itération).

**Exemple.**
Voici deux fonctions qui calculent $n!$ (factorielle $n$) si $n$ est un entier naturel.
On veut prouver leur correction.

**Fonction avec `for`**

```
def Factorielle(n:int)->int:
    p , a = n , n
    for i in range(n-1):
        a = a - 1
        p = p * a
    return p
```

**Fonction avec `while`**

```
def Factorielle(n:int)->int:
    p , a = 1 , 0
    while a < n:
        a = a + 1
        p = p * a
    return p
```

• **Fonction avec `for` :**

Terminaison : $n-1-i$ est un variant de boucle, c'est une suite d'entiers strictement décroissante et positive.

Correction : Un invariant de boucle est : $\mathcal{P}(i)$ : "$a_i = n-i-1$ et $p_i = \dfrac{n!}{(n-2-i)!}$".

Il suffit ensuite de faire la preuve par récurrence.

• **Fonction avec `while` :**

Terminaison : $n-1-i$ est un variant de boucle, c'est une suite d'entiers strictement décroissante et positive.

Correction : Un invariant de boucle est : $\mathcal{P}(i)$ : "$a_i = i$ et $p_i = i!$".

Il suffit ensuite de faire la preuve par récurrence.

---

**Exemple.** On donne deux fonctions : la première détermine le maximum des éléments d'une liste de flottants et la seconde trie une liste de flottants avec le **tri par sélection** (*cf. paragraphe sur les tris*).

**Maximum des éléments d'une liste**

```
def maximum(L:list)->float:
  M=L[0]
  for i in range(1,len(L)):
    if L[i]>M:
      M=L[i]
  return M
```

**Tri par sélection d'une liste**

```
def tri_par_selection(L:list)->list:
  for i in range(len(L)):
    ind_min=i
    for j in range(i+1,len(L)):
      if L[j]<L[ind_min]:
        ind_min=j
    L[i],L[ind_min]=L[ind_min],L[i]
  return L
```

• **Maximum des éléments d'une liste**

Terminaison :
`len(L)-1-i` est un variant de boucle.

Correction :
Un invariant de boucle est :

$\mathcal{P}(i)$ : "$M_i$ est le maximum de la liste `L[:i]`"

Initialisation :
Première boucle ($i=1$) on a bien $M_1$ maximum de la liste `[L[0]]`.

Transmission : $\mathcal{P}(i) \Rightarrow \mathcal{P}(i+1)$

On suppose $\mathcal{P}(i)$, $M_i$ est le maximum de `L[:i]`, on compare $M_i$ à l'élément `L[i]` et on garde le maximum donc $M_{i+1}$ est le maximum de `L[:i+1]`. Ainsi $\mathcal{P}(i+1)$ est vraie.

Conclusion : La dernière étape $\mathcal{P}(n-1)$ donne $M_{n-1}$ maximum de la tranche `L[:n-1]=L` et l'algorithme est correct.

• **Tri par sélection d'une liste**

La boucle sur `j` est analogue à celle sur le maximum. On ne prouve que la correction et la terminaison de la boucle sur `i`.

Terminaison : `len(L)-1-i` est un variant de boucle.

Correction : Un invariant de boucle est :

$\mathcal{P}(i)$ : `L[:i]` est triée et les éléments de `L[i:]` sont supérieurs à ceux de `L[:i]`"

Initialisation : Première boucle : l'élément le plus petit d'indice `ind_min` a été permuté avec `L[0]` donc les autres éléments sont plus grands !

Transmission : $\mathcal{P}(i) \Rightarrow \mathcal{P}(i+1)$
La tranche `L[:i]` est triée et `L[i:]` ne contient que des éléments supérieurs à ceux de `L[:i]`.
Or l'élément le plus petit de `L[:i]` est ramené à la position `i` de la liste, celui-ci est plus grand que tout les éléments de `L[:i]` donc `L[:i+1]` est triée. De plus la liste `L[i+1:]` ne contient plus que des éléments plus grands que `L[i]` donc que ceux de `L[:i+1]`. On a prouvé $\mathcal{P}(i+1)$.

Conclusion : La dernière étape est $\mathcal{P}(n-1)$ : la liste `L[:n-1]` est triée et le dernier élément est plus grand !
Donc `L` est triée et L'algorithme est donc correct.

# COMPLEXITÉ

Le programme de CPGE est clair :
- On aborde la notion de complexité temporelle dans le pire cas en ordre de grandeur.
- On peut, sur des exemples, aborder la notion de complexité en espace.

## QUELQUES ORDRES DE GRANDEUR

Le tableau suivant présente le $n$ maximal tel qu'un algorithme nécessitant $f(n)$ opérations élémentaires pour s'exécuter sur l'entrée $n$ en moins d'une minute (en supposant $10^9$ **opérations élémentaires par seconde**).

| $f(n)$ | $\ln(n)$ | $\sqrt{n}$ | $n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ | $n^n$ |
|---|---|---|---|---|---|---|---|---|
| $n_{\max}$ | très très gros | $3,6\times10^{21}$ | $6\times10^{10}$ | 244948 | 3914 | 35 | 13 | 10 |

Voici le temps d'exécution pour un **problème de taille** $n=10^6$ sur un ordinateur à $10^9$ **opérations par seconde** :

| | | | |
|---|---|---|---|
| $\mathcal{O}(1)$ | Temps constant | 1 ns | Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare ! |
| $\mathcal{O}(\ln(n))$ | Logarithmique | 1 $\mu$s | En pratique, cela correspond à une exécution quasi instantanée. Bien souvent, pour des algorithmes dichotomiques c'est $\log_2(n)$ que l'ont voit apparaître. |
| $\mathcal{O}(n)$ | Linéaire | 1 ms | Le temps d'un tel algorithme ne devient supérieur à 1 min que pour des données de taille comparable à celle des mémoires vives actuelles. |
| $\mathcal{O}(n\ln(n))$ | Quasi-linéaire | 10 ms | Le temps d'exécution est légèrement supérieur au cas linéaire. |
| $\mathcal{O}(n^2)$ | Quadratique | 1/4h | Cette complexité est acceptable pour $n<100$ mais pas au delà. |
| $\mathcal{O}(n^k)$ | Polynômial | 30 ans | Il n'est pas rare de voir des complexités en $\mathcal{O}(n^3)$ ou $\mathcal{O}(n^4)$. |
| $\mathcal{O}(2^n)$ | Exponentielle | $10^{30000}$ ans | Un algorithme de telle complexité est impraticable, sauf pour des données très petites. |

## DÉFINITION ET NOTATIONS DE LANDAU

▶ La **complexité** est la mesure de l'efficacité d'un programme pour un type de ressources :
- **complexité temporelle** : temps de calcul CPU (processeur) : **nombre d'opérations élémentaires** réalisés par le processeur, ce qui est lié directement au temps de calcul.
- **complexité spatiale** : espace mémoire nécessaire : mémoire RAM ou sur disque dur ...

▶ La **complexité temporelle asymptotique** est l'étude de l'ordre de grandeur de la complexité pour des entrées de grande taille « dans le pire des cas » (lorsque la variable entrée nécessite le maximum d'opérations dans chaque étape de l'algorithme).

**Remarque. En pratique, la complexité temporelle est plus importante que la complexité spatiale.**

▶ On exprime la complexité en fonction de $n$, la taille de l'entrée (*longueur de la liste, de la chaîne de caractères, de l'entier lui-même ou bien du nombre de chiffres de l'entier entré*).

▶ Les **notations de Landau** ($\mathcal{O}(\ln(n))$, $\mathcal{O}(n)$, $\mathcal{O}(n^k)$ où $k\in\mathbb{N}$, $\mathcal{O}(2^n)$) sont les mieux adaptées pour l'exprimer.

▶ On note $C(n)$ la complexité d'un algorithme (nombre d'opérations élémentaires effectuées dans l'algorithme).

## COMPLEXITÉ DES COMMANDES DE BASE

▶ Toute ligne de code écrite comporte des commandes élémentaires et chaque commande a sa propre complexité.

▶ En général, leur exécution coûte $\mathcal{O}(1)$ en complexité. (pas toutes : voir ci-après pour les listes)

| Opérations | syntaxe | Complexité |
|---|---|---|
| d'affectation | `a=2` | $\mathcal{O}(1)$ |
| sur les entiers | `+,-,*,//,%` | $\mathcal{O}(1)$ |
| sur les flottants | `+,-,*,/,**` | $\mathcal{O}(1)$ |
| sur les booléens | `not,or,and` | $\mathcal{O}(1)$ |
| de comparaison | `==,!=,<,>,<=,>=` | $\mathcal{O}(1)$ |
| d'affichage | `print()` | $\mathcal{O}(1)$ |

**Remarque.** Ce n'est pas parce que la syntaxe d'une instruction est courte qu'elle a une complexité en $\mathcal{O}(1)$ !!!

**Remarque.** Pour de très grands entiers $n$, le temps d'exécution de `*,//,%` est légèrement plus élevé que `+,-`. Néanmoins dans la pratique on ne compte qu'une « opération » pour chacune de ces commandes.

**Remarque.** De même, le temps d'exécution de `**` sur les flottants est plus élevé que celui des autres opérations (mais raisonnable grâce à l'algorithme d'exponentiation rapide).

### ▶ Cas particulier des commandes sur les listes

En Python, il est tout à fait possible faire des opérations sur tout une liste `L` à l'aide d'un seule commande. Il faut avoir à l'esprit que ce ne sont pas des opérations élémentaires (*elles ont chacune leur propre complexité*), et certaines cachent un travail important pour le processeur !

| Opération | Exemple | Complexité |
|---|---|---|
| Accès direct | `L[5]` | $\mathcal{O}(1)$ |
| Affectation | `L[-1]=10` | $\mathcal{O}(1)$ |
| Longueur | `len(L)` | $\mathcal{O}(1)$ |
| Ajout en fin de liste | `L.append(3.14)` | $\mathcal{O}(1)$ |
| Suppression en fin de liste | `L.pop()` | $\mathcal{O}(1)$ |
| Suppression d'un élément | `L.pop(7)` | $\mathcal{O}(n)$ (*commande hors-programme*) |
| Concaténation | `L1+L2` | $\mathcal{O}(n_1+n_2)$ (longueurs des deux listes) |
| Concaténation (bis) | `L+[x]` | $\mathcal{O}(n)$ où $n=$ `len(L)` |
| Extraction de tranche | `L[1:100]` | $\mathcal{O}(n)$, où $n=$ longueur de la tranche. |
| Copie profonde | `M=L[:]` ou `M=L.copy()` | $\mathcal{O}(n)$, où $n=$ `len(L)`. |
| Répétition | `[0]*n` | $\mathcal{O}(n)$ |
| Création par compréhension | `[k**2 for k in range(n)]` | $\mathcal{O}(n)$ si l'expression est évaluée en temps constant |
| Recherche de `x` dans `L` | `x in L` | $\mathcal{O}(n)$, où $n$ est la longueur de la liste créée. |
| Recherche de `x` dans `D` | `x in D` | $\mathcal{O}(1)$, où `D` est un **dictionnaire**. |

**Remarque.**

▶ On préfèrera donc `L.append(x)` à `L+=[x]` et on évitera d'utiliser `L.pop(i)` alors que `L.pop()` on peut.

▶ Idem pour les tranches, ou les copies de listes, on évite si on peut faire autrement.
(Dans les algorithmes de tri par exemple.)

▶ Les commandes analogues sur les chaînes de caractères ou dictionnaires ont la même complexité.
Attention à l'exception de la commande `x in D` de complexité $\mathcal{O}(1)$ grâce aux fonctions de hachages (voir cours de PSI).

## COMPLEXITÉ SPATIALE

▶ C'est la mesure de l'espace mémoire maximal occupé durant l'exécution de l'algorithme.

▶ On définit l'unité d'espace mémoire comme la taille d'une structure de données particulières.

▶ On mesure la complexité en espace par le nombre $n$ d'unités de mémoire occupées lors de l'exécution de l'algorithme.

| Tableau de nombres de longueur | Unité d'espace mémoire | Espace mémoire occupé par le tableau |
|---|---|---|
| $n$ | taille d'un entier/flottant | $\mathcal{O}(n)$ |
| $5n$ | taille d'un entier/flottant | $\mathcal{O}(n)$ |
| $m \times n$ | taille d'un flottant/flottant | $\mathcal{O}(n \times m)$ |

**Exemple.** Une image en 4K est une image rectangulaire de 4000 pixels sur 8000 pixel. Chaque pixel a un code RGB stocké sur un triplet d'entiers entre 0 et 255.

La complexité spatiale de cette image (en bits) est : $4000 \times 8000 \times 3 \times 8 = 7,68 \times 10^8$ bits.

La complexité spatiale de cette image (en Mo) est : $4000 \times 8000 \times 3 : 10^6 = 7,68 \times 10^8 = 96$ Mo.

## EXEMPLES AVEC ALGORITHMES ITÉRATIFS

- Un premier exemple montrant 4 façons de calculer la même quantité avec différentes complexités :

| **Calcul mathématique** | **Code Python** | **Complexité** |
|---|---|---|
| $S_n = \dfrac{n \times 2 \times (1+n) \times n}{2}$<br>$S_n = n^2(n+1)$ | `n = 100`<br>`Res = n*2*(1+n)*n/2`<br>`print(Res)` | $\mathcal{O}(1)$ |
| $S_n = \sum_{i=1}^{n} 2 \times n \times i$<br>$S_n = 2n \times \sum_{i=1}^{n} i$<br>$S_n = 2n \times \dfrac{n(n+1)}{2}$<br>$S_n = n^2(n+1)$ | `Res = 0`<br>`n = 100`<br>`for i in range(1,n+1):`<br>`  # Somme des termes de 1 à n`<br>`  Res = Res + 2*n*i`<br>`print(Res)` | $\mathcal{O}(n)$ |
| $S_n = \sum_{i=1}^{n}\sum_{j=1}^{n}(i+j)$<br>$S_n = \sum_{i=1}^{n}\sum_{j=1}^{n} i + \sum_{i=1}^{n}\sum_{j=1}^{n} j$<br>$S_n = \sum_{i=1}^{n} n \times i + \sum_{i=1}^{n} \dfrac{n(n+1)}{2}$<br>$S_n = n \times \frac{n(n+1)}{2} + n \times \frac{n(n+1)}{2}$<br>$S_n = n^2(n+1)$ | `Res = 0`<br>`n = 100`<br>`for i in range(1,n+1):`<br>`  # Somme des termes de 1 à n`<br>`  for j in range(1,n+1):`<br>`  Res = Res + i + j`<br>`print(Res)` | $\mathcal{O}(n^2)$ |
| $S_n = \sum_{i=1}^{n} n \times i + \sum_{j=1}^{n} n \times j$<br>$S_n = n \times \frac{n(n+1)}{2} + n \times \frac{n(n+1)}{2}$<br>$S_n = n^2(n+1)$ | `Res = 0`<br>`n = 100`<br>`for i in range(1,n+1):`<br>`  Res += n*i`<br>`for j in range(1,n+1):`<br>`  Res += n*j`<br>`print(Res)` | $\mathcal{O}(n)$ |

## EXEMPLES AVEC ALGORITHMES RÉCURSIFS

• Voici un exemple simple : $u_0 = 1$ et $\forall n \geqslant 1, u_{n+1} = \begin{cases} u_n + 1 \text{ si } u_n < 1 \\ \dfrac{u_n}{2} \text{ sinon} \end{cases}$.

| Code Python | Complexité de l'algorithme |
|---|---|
| `def rec(n):`<br>`  if n==0:`<br>`  return 1`<br>`  else:`<br>`  U = rec(n-1)`<br>`  if U < 1:`<br>`  U = U + 1`<br>`  else:`<br>`  U = U / 2`<br>`  return U` | Soit $C(n)$ la complexité de `rec` à l'appel $n$ :<br>$C(0) = 1$<br>$C(n+1) = C(n) + 5$<br>(Suite arithmétique)<br>$C(n) = 5n + 1$<br>$\Rightarrow$ Complexité linéaire en $\mathcal{O}(n)$ |
| `def rec(n):`<br>`  if n==0:`<br>`  return 1`<br>`  else:`<br>`  if rec(n-1) < 1:`<br>`  U = rec(n-1) + 1`<br>`  else:`<br>`  U = rec(n-1) / 2`<br>`  return U` | Soit $C(n)$ la complexité de `rec` à l'appel $n$ :<br>$C(0) = 1$<br>$C(n+1) = 2C(n) + 5$<br>(Suite arithmético-géométrique)<br>$C(n) + 5 = 2^n(C(0) + 5) \Leftrightarrow C(n) = 6 \times 2^n + 5$<br>$\Rightarrow$ Complexité exponentielle en $\mathcal{O}(2^n)$ |

**Remarque.**

▶ Tout commande compte pour une ou plusieurs « opération(s) » (même l'affectation `=`).

▶ Le but n'est pas d'évaluer exactement le nombre d'opérations mais de savoir si celui-ci est en $\mathcal{O}(1)$, $\mathcal{O}(\ln(n))$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, ou $\mathcal{O}(2^n)$.

▶ Pour les algorithme récursifs, l'idée est surtout de savoir combien de fois ils font appel à eux-même à l'étape précédente mais aussi combien d'opérations sont faites entre deux appels successifs.

## EXEMPLES AVEC ALGORITHMES DICHOTOMIQUES

• Voici un exemple avec l'algorithme d'exponentiation rapide. On évalue la complexité à l'étape $2^n$ (et pas $n$).

| Code Python | Complexité de l'algorithme |
|---|---|
| `def Quick_exp(a,n):`<br>`  if n==1:`<br>`  return a`<br>`  if n%2==0:`<br>`  y=Quick_exp(a,n//2)`<br>`  return y*y`<br>`  else:`<br>`  y=Quick_exp(a,(n-1)//2)`<br>`  return y*y*a` | Soit $C(2^n)$ la complexité de `rec` à l'appel $2^n$ :<br>$C(0) = 1$<br>$C(2^{n+1}) = C(2^n) + 7$<br>(Suite arithmétique)<br>$C(2^n) = 7n + 1$<br>$C(n) = 7\log_2(n) + 1$<br>$\Rightarrow$ Complexité logarithmique en $\mathcal{O}(\ln(n))$ |

**Remarque.**

▶ L'idée est compter ici le nombre total d'appels $p$ à la fonction en fonction de $n$.

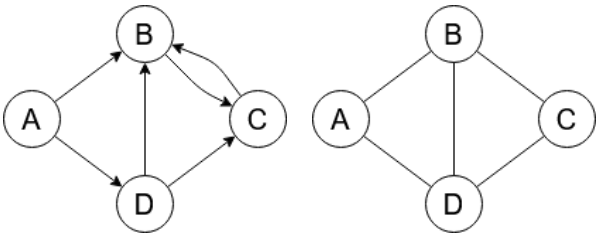
▶ L'algorithme s'appelle jusqu'à ce que $\frac{n}{2^p} \approx 1$ soit $p \approx \log_2(n)$ donc la complexité est en $\mathcal{O}(\ln(n))$.

# GRAPHES : GÉNÉRALITÉS

## LES TYPES DE GRAPHES

### ▶ Graphes non orientés

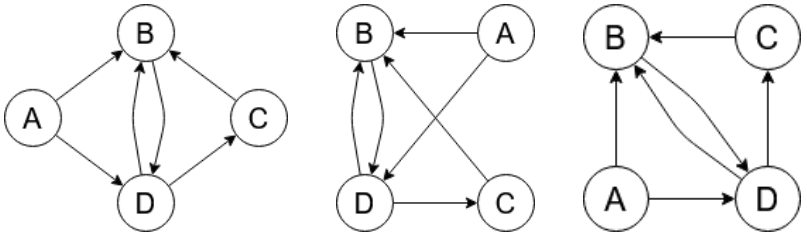
• Un **graphe non orienté d'ordre** $n$ noté $G$ est un ensemble de $n$ points ou **sommets**, reliés entre eux ou non par des lignes appelées **arêtes**.
• L' **ordre** d'un graphe est donc son nombre de sommets.
• On note souvent $G = (S, A)$ où $S$ est la liste des sommets et $A$ la liste des arêtes.
• Les sommets peuvent avoir des noms, être numérotés, etc...



**Graphe orienté** **Graphe non orienté**

### ▶ Graphes orientés

• Un **graphe orienté d'ordre** $n$ noté $G$ est un ensemble de $n$ points ou **sommets**, reliés entre eux ou non par des flèches appelées **arc**.
• Un sommet peut être relié à lui-même par une **boucle**.
• Un graphe sans boucle est appelé graphe **simple**.
• Le même graphe admet une infinité de représentations.
• Les arcs sont donc des arêtes orientées.

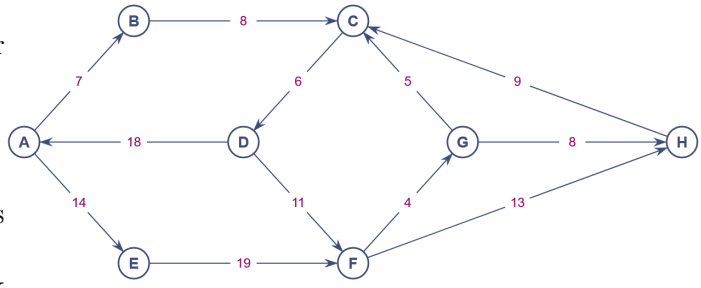


**Trois représentations d'un même graphe**

### ▶ Graphes pondérés

Un **graphe pondéré** est un graphe, orienté ou non, pour lequel chaque arête possède un **poids**.

**Remarque.**
• Généralement le poids est un nombre réel positif ou nul.
• On peut mettre le poids à 0 pour signifier que deux sommets ne sont pas reliés.
• Le poids peut représenter le temps de parcours entre deux sommets, le prix du parcours...

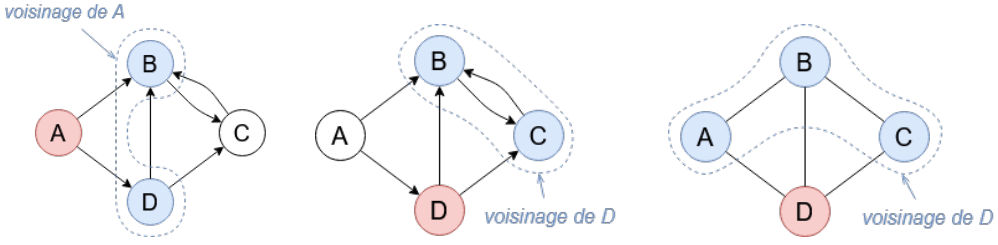


**Un exemple de graphe pondéré**

## LEXIQUE DES GRAPHES

• **ordre** d'un graphe est le nombre de sommets de ce graphe.
• Un sommet $B$ est **adjacent** à sommet $A$, ou $B$ est un **voisin** de $A$ lorsque que $A$ est relié à $B$ par une arête/un arc.
• Un sommet qui n'est adjacent à aucun autre est dit **isolé**.

**Exemple. Trois voisinages différents dans des graphes orientés ou non**



• Le **degré** $d(s)$ d'un sommet $s$ est est le nombre d'arêtes/d'arcs (quel que soit leur sens) auxquels il est relié (une boucle compte deux fois).

Pour un graphe orienté, on définit le **degré entrant** et le **degré sortant**.
- On note $d_-(s)$ le **degré entrant** du sommet $s$ : nombre d'arcs ayant $s$ comme extrémité finale.
- On note $d_+(s)$ le **degré sortant** du sommet $s$ : nombre d'arcs ayant $s$ comme extrémité initiale.
- Le **degré** du sommet $s$ est donc $d(s) = d_+(s) + d_-(s)$

Dans un graphe, la somme des degrés de chaque sommet est égale au double du nombre d'arêtes.

• Un graphe est **complet** lorsque toutes les sommets sont deux à deux adjacentes entre eux.

• Dans un graphe non orienté, une **chaine** de **longueur** $n$ est une séquence finie de sommets reliés entre eux par $n$ arêtes. (*L'extrémité de chacune est l'origine de la suivante, sauf pour la dernière.*)
• Dans un graphe orienté, on parle de **chemin** et il suit le sens des flèches.

• Un chemin ou une chaîne est qualifié de **simple** s'il n'emprunte pas deux fois la même arête.
• Un **chemin** est **élémentaire** s'il ne passe pas deux fois par le même sommet.

• Si une chaîne/un chemin possède le même sommet de départ et d'arrivée, on dit qu'elle/il est **fermé(e)**.
• Un chemin simple fermé s'appelle une **cycle/circuit**.

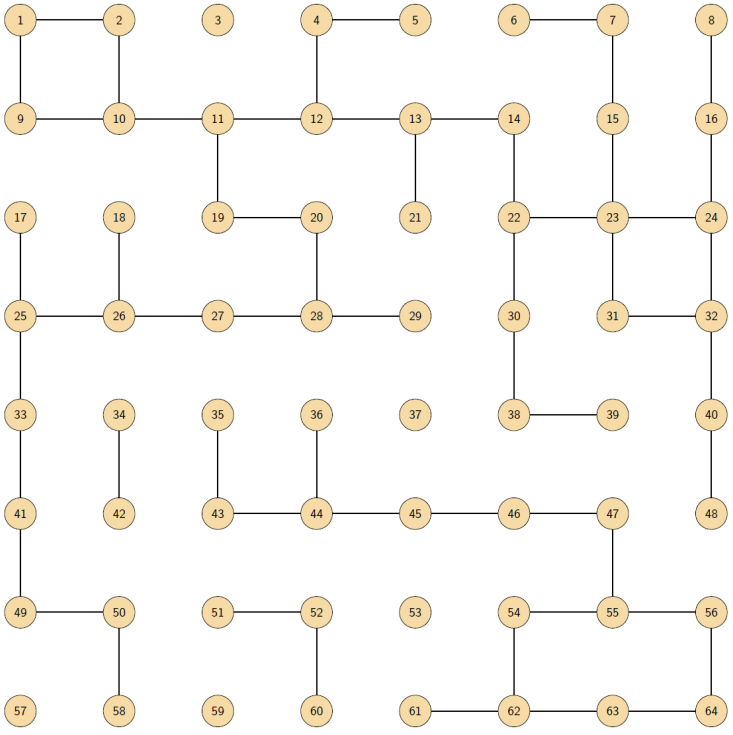
• Pour un graphe pondéré, la longueur/**poids** d'un(e) chaîne/chemin est la somme des poids qui la compose.

• La **distance** entre deux sommets d'un graphe est la longueur du chemin/circuit le plus court (s'il y en a un) reliant ces deux sommets.

• Un **graphe non orienté** est **connexe** s'il existe une chaîne entre chaque paire de sommets distincts.

• Pour les **graphes orientés** on parle de : (*mais il semble que ce n'est pas au programme*)
– **graphe orienté connexe** si le graphe non orienté associé est connexe (graphe obtenu en ne tenant pas compte du sens des arêtes).
– **graphe orienté fortement connexe** si pour toute paire $(A, B)$ de sommets, il existe un chemin de $A$ vers $B$ et un chemin de $B$ vers $A$.

**Exemple.** Soit $G$ le graphe ci-contre.




• $G$ est non orienté d'ordre 64.

• Le sommet 3 est isolé.

• Le sommet 11 est de degré 3.

• $1-9-10-11-19-20-28-29$ est une chaine de longueur 7.

• $54-55-56-64-63-62-54$ est un cycle.

• $1-2-10-9-1-2-10-9-1$ n'est pas un cycle.

• La distance entre 1 et 8 est de 11

• $G$ n'est pas connexe.

• $G$ admet 8 composantes connexes.

(*On dit que deux sommets sont dans la même « composante connexe » lorsqu'il existe un chemin reliant ses deux sommets.*
*Le graphe est ainsi séparé en un certain nombre de composantes connexes distinctes.*)

## MODÉLISATION DES GRAPHES

### ▶ Théorique

Soit $S = \{s_1, \dots, s_n\}$ l'ensemble des sommets d'un graphe $G$. On appelle $A$ l'ensemble :

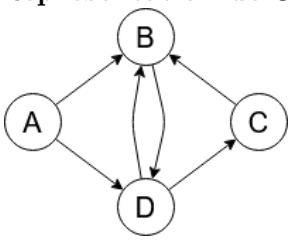
— Des arêtes d'un graphe non orienté tel que $A = \{\{s_i, s_j\}, s_i \in S, s_j \in S\}$

— Des arcs d'un graphe orienté tel que $A = \{(s_i, s_j), s_i \in S, s_j \in S\}$

On note $G = (S, A)$ le graphe composé de ces sommets et arêtes/arcs.
On note $G = (S, A, \omega)$ un graphe pondéré tel que chaque arc/arête a un poids $\omega(s_i, s_j)$.

**Remarque.**
• On a besoin d'une liste de noms de sommets et d'une liste des arêtes/arcs/sommets voisins.
• Notez que pour $\{s_i, s_j\}$ (arête) l'ordre ne compte pas alors que pour $(s_i, s_j)$ (arc) l'ordre compte.

**Représentation de $G$**




**Matrice d'adjacence :**

```
G = [
    [0,1,0,1],
    [0,0,0,1],
    [0,1,0,0],
    [0,1,1,0]
    ]
```

**Dictionnaire d'adjacence :**

```
G = {
    'A': ['B','D'],
    'B': ['D'] ,
    'C': ['B'] ,
    'D': ['B','C']
    }
```

### ▶ Matrice d'adjacence

Soit un graphe de $n$ sommets, d'indices $0, \dots, n-1$.
• La **matrice d'adjacence** de ce graphe est un tableau `G` à deux dimensions, de taille $n \times n$, contenant des booléens `G[i][j]` (0 ou 1) indiquant si il y a adjacence entre les sommets d'indices $i$ et $j$.
(Attention : `G[i][j]` pour un arc de $i$ vers $j$ : $i \to j$ et `G[j][i]` pour un arc de $j$ vers $i$ : $j \to i$)

• Pour les graphes pondérés, on parle de **matrice des distances**, où le coefficient `G[i][j]` vaut le poids de l'arête/arc reliant le somme $s_i$ au sommet $s_j$.

**Remarque. Inconvénient de la matrice d'adjacence :**
• La dimension de la matrice est égale à $n \times n$, ce qui peut représenter un important espace en mémoire.
• Pour obtenir les voisins d'un sommet, il faut une utiliser une fonction de coût d'ordre $O(n)$.
• Il faut stocker la correspondance noms/indice des sommets dans un dictionnaire à part.
Le graphe `G` peut être alors définit par le couple `(D,M)` où `D` est le dictionnaire des noms des sommets et `M` la matrice adjacence.

### ▶ Dictionnaire d'adjacence

Soit un graphe de $n$ sommets ayant des noms distincts.
Le **dictionnaire d'adjacence** de ce graphe est un dictionnaire où chaque élément est un sommet et la clef associée est la liste des sommets adjacents à celui-ci.

**Remarque.**
• On n'a pas besoin de stocker les noms des sommets dans un dictionnaire à part.
• S'il y a peu d'arête, la taille du dictionnaire est bien inférieur à celle de la matrice d'adjacence.

• Pour un graphe pondéré le **dictionnaire d'adjacence** est un dictionnaire où chaque élément est un sommet et la clef associée est la liste des sommets adjacents à celui-ci couplé avec le poids de l'arête correspondante.

## ALGORITHMES CLASSIQUES SUR LES GRAPHES

### ▶ Ordre d'un graphe (matrice)

```
def Ordre(G):
  return len(G)
```

### ▶ Ordre d'un graphe (dictionnaire)

```
def Ordre(G):
  return len(G)
```

---

### ▶ Degré sortant d'un sommet (matrice)

```
def dplus(G,s):
    dp=0
    for j in range(Ordre(G)):
        if G[s][j]!=0:
            dp+=1
    return dp
```

### ▶ Degré sortant d'un sommet (dictionnaire)

```
def dplus(G,s):
    return len(G[s])
```

**Remarque.**
C'est aussi le degré si le graphe n'est pas orienté.

---

### ▶ Degré entrant d'un sommet (matrice)

```
def dmoins(G,s):
    dm=0
    for i in range(Ordre(G)):
        if G[i][s]!=0:
            dm+=1
    return dm
```

### ▶ Degré entrant d'un sommet (dictionnaire)

```
def dmoins(G,s):
    dm=0
    for g in G:
        if s in G[g]:
            dm+=1
    return dm
```

---

### ▶ Voisins d'un sommet (matrice)

```
def Voisins(G,s):
    V=[]
    for j in range(len(L)):
        if G[s][j]!=0:
            V.append(j)
    return V
```

### ▶ Voisins d'un sommet (dictionnaire)

```
def Voisins(G,s):
    return G[s]
```

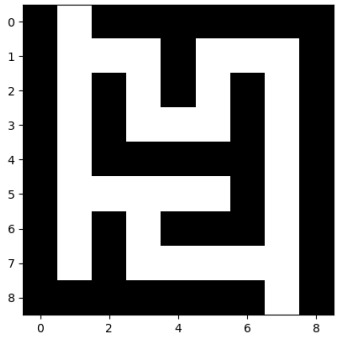
---

### ▶ Voisins dans un contexte concret de labyrinthe

On considère un labyrinthe modélisé comme une **matrice** (une liste de listes). Les **cases blanches** sont codées 1 et les **cases noires** (murs) sont codées 0. On circule dans le labyrinthe uniquement sur les cases blanches. L'**entrée** est en haut à gauche et la **sortie** en bas à droite.

Les voisines d'une case blanche, notée `case`, sont les cases blanches adjacentes (en haut, en bas, à droite, à gauche). Il y en a 4 au maximum, en faisant attention aux « bords ».

Exemple de labyrinthe $9 \times 9$.




Matrice $9 \times 9$ associée

```
M=[[0,1,0,0,0,0,0,0,0],
   [0,1,1,1,0,1,1,1,0],
   [0,1,0,1,0,1,0,1,0],
   [0,1,0,1,1,1,0,1,0],
   [0,1,0,0,0,0,0,1,0],
   [0,1,1,1,1,1,0,1,0],
   [0,1,0,1,0,0,0,1,0],
   [0,1,0,1,1,1,1,1,0],
   [0,0,0,0,0,0,0,1,0]]
```

```
def Voisins(M,case):
    V,n,p=[],len(M),len(M[0])
    i,j=case
    if i>0 and M[i-1][j]==1:
        V.append((i-1,j))
    if i<n-1 and M[i+1][j]==1:
        V.append((i+1,j))
    if j>0 and M[i][j-1]==1:
        V.append((i,j-1))
    if j<p-1 and M[i][j+1]==1:
        V.append((i,j+1))
    return V
```

# GRAPHES : PARCOURS DE GRAPHES

## GRAPHES : PARCOURS EN LARGEUR

### ▶ Principe

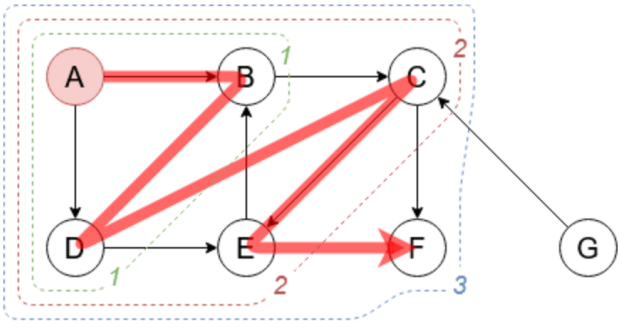
• Dans le **parcours en largeur**, on visite les sommets en « cercle concentriques » autour du sommet de départ.

• Le principe peut se décrire ainsi : Les frères avant les fils, une génération après l'autre.

### ▶ Exemple

• Début : Sommet $A$ (père).
fils de $A$ : $B$ et $D$ (distance 1)
• fils de $B$ : $C$ (distance 2)
fils de $D$ : $E$ (distance 2)
• fils de $C$ : $F$ (distance 3)
fils de $E$ : $F$ déjà parcouru.
• $G$ n'est pas atteignable.

**Parcours :**
$A$
$A-B-D$
$A-B-D-C$
$A-B-D-C-E$
$A-B-D-C-E-F$




### ▶ File et format deque

La liste des sommets à visiter ressemble une file d'attente. On va donc utiliser le concept de **file** où c'est le premier élément (celui qui a été inscrit en premier dans la liste) qui va être traité d'abord. Il y a deux opérations :

— mettre un élément dans la file (en dernier donc) : « enfiler » avec `L.append(x)`

— supprimer un élément de la file (le premier donc) : « défiler » avec `L.pop(0)`

Selon le programme officiel, aucune connaissance n'est théoriquement exigible sur les files et les modules nécessaires. Néanmoins il faut avoir souligné l'utilité de tels objets pour le parcours de graphes (en largeur).

| | |
|---|---|
| Importer le module | `from collections import deque` |
| Créer une file | `file=deque(['A','B','D'])` |
| Enfiler | `file.append('F')` |
| Défiler le premier arrivé | `file.popleft()` |

**Remarque.**
• L'essentiel est de comprendre que la variable `file` créé avec la commande `deque` se gère comme une liste sauf que la commande `file.popleft()` a une complexité en $\mathcal{O}(1)$ tandis que celle de `L.pop(0)` est en $\mathcal{O}(\texttt{len(L)})$.
• Il existe également des commandes pour enfiler à gauche ou défiler.

### ▶ Algorithme itératif

• `G` = matrice/dictionnaire d'adjacente et `dep` l'indice/le nom du sommet de départ.

• `Parcours` = liste des sommets parcourus dans l'ordre via le parcours en largeur.

• `Vu` = dictionnaire des sommets vus.

• `file` = liste d'attente des sommets à voir.

• `Voisins(G,s)` = fonction qui renvoie la liste des sommets voisins de `s` dans le graphe `G`.

```
1  def Parcours_Largeur(G,dep)->list:
2      """Parcours en largeur itératif avec file"""
3      Parcours=[]
4      Vu={}
5      file=[dep] # ou file=deque([dep])
6      while len(file)!=0:
7          s=file.pop(0) # ou s=file.popleft()
8          if s not in Vu:
9              Vu[s]=True
10             Parcours.append(s)
11             for v in Voisins(G,s):
12               if v not in Vu:
13                   file.append(v)
14     return Parcours
```

## GRAPHES : PARCOURS EN PROFONDEUR

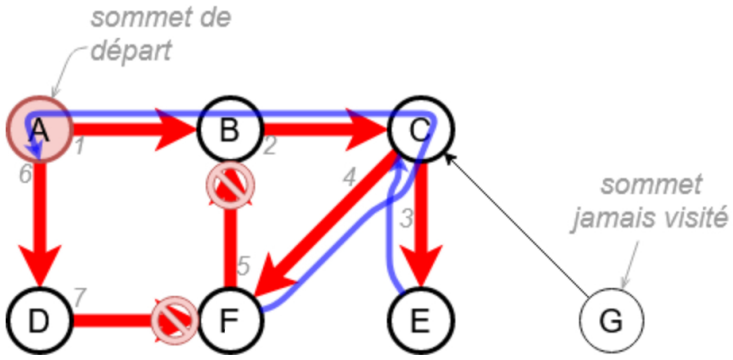
### ▶ Principe

• Le **parcours en profondeur** d'un graphe à partir d'un sommet consiste à suivre les arêtes arbitrairement, en marquant les sommets déjà visités pour ne pas les visiter à nouveau.

• La descendance avant les frères. On avance le plus possible et on recule quand on est bloqué. On remonte alors au premier ancêtre qui a un fils qui n'a pas été visité.

### ▶ Exemple

• Début : Sommet $A$ (père).
Fils de $A$ : $B$ et $D$ dans cet ordre.
• Fils de $B$ : $C$
• Fils de $C$ : $E$ et $F$ dans cet ordre.
• Fils de $E$ non vus : $\emptyset \Rightarrow$ bloqué $\Rightarrow$ remonte à $C$.
Second fils de $C$ : $F$
• Fils de $F$ non vu : $\emptyset \Rightarrow$ bloqué $\Rightarrow$ remonte à $A$.
Second fils de $A$ : $D$.
• Fils de $D$ non vu : on remonte à $A$.
• Fils de $A$ non vus : $\emptyset \Rightarrow$ on a fini !




Parcours de $G$ à partir de $A$ : `[A,B,C,E,F,D]`.

### ▶ Algorithme récursif

• `G` = matrice/dictionnaire d'adjacence.
• `S` = indice/le nom du sommet de départ.
• `Parcours` = liste des sommets parcourus dans l'ordre du parcours en profondeur.
• `Vu` = dictionnaire des sommets vus.
• `Voisins(G,S)` = fonction qui renvoie la liste des sommets voisins de `s` dans le graphe `G`.

```
def Parcours_Profondeur(G,S,Vu,Parcours):
  """ Parcours en profondeur récursif """
  if S not in Vu:
    Vu[S]=True
    Parcours.append(S)
    for v in Voisins(G,S):
      if v not in Vu:
        Parcours_Profondeur(G,v,Vu)

Vu,Parcours={},[] # Initialisation
Parcours_Profondeur(G,'A',Vu,Parcours)
print(Parcours) # Affichage
```

**Remarque.**
• Le dictionnaire `Vu` et la liste `Parcours`, initialement vides, sont modifiés par effet de bord à chaque appel de `Parcours_Profondeur`.
• On atteint vite la limite de la récursivité (profondeur maximale de récursivité en Python).
• Il existe une fonction alternative itérative, à l'aide d'une pile.

### ▶ Pile et format `deque`

Une **pile** est un concept informatique basée sur le principe « dernier arrivé, premier sorti ». Pour le parcours en profondeur, à chaque fois qu'on visite un sommet pour la première fois, on rajoute à la pile tous ses voisins non encore visitée et on continue tant que la pile n'est pas vide. Il faut donc savoir créer une pile et :

- rajouter un élément en haut de la pile : « empiler » avec `L.append(x)`
- enlever le dernier élément de la pile : « dépiler » avec `L.pop()`

Selon le programme officiel, aucune connaissance n'est théoriquement exigible sur les piles et les modules nécessaires. Néanmoins il faut avoir souligné l'utilité de tels objets pour le parcours de graphes (en profondeur).

**Remarque.**
• La syntaxe des piles est la même que celle des listes (sauf création avec `deque`).
• Les piles sont des structures faciles à gérer mais toute manipulation élaborée nécessitera énormément d'empilages et de dépilages.

| | |
|---|---|
| Importer le module | `from collections import deque` |
| Créer une pile | `pile=deque(['A','B','D'])` |
| Empiler | `pile.append('F')` |
| Dépiler le dernier arrivé | `pile.pop()` |

### ▶ Algorithme itératif

• On retrouve les variables `Vu`, `Parcours`, `G` et `pile` est la pile/liste des sommets à visiter.

Tant que la pile n'est pas vide,

- on dépile le dernier sommet de la pile,
- si jamais ce sommet n'a pas été visité on l'ajoute à la liste des sommets visités,
- puis on fait la liste de ses voisins,
- enfin on ajoute à la pile ceux que l'on a pas encore visité.

```
def Parcours_Profondeur(G,dep)->list:
    """parcours en profondeur avec pile"""
    Parcours=[]
    Vu={}
    pile=[dep] # ou pile=deque([dep])
    while len(pile)!=0:
        s=pile.pop()
        if s not in Vu:
            Vu[s]=True
            Parcours.append(s)
            for v in Voisins(G,s):
              if v not in Vu:
                  pile.append(v)
    return Parcours
```

**Remarque.**

• Cet algorithme ne renvoie pas exactement le même parcours en profondeur que la version récursive : en effet dans la boucle `for v in Voisins(G,s)`, l'instruction `pile.append(v)` ajoute les sommets par ordre croissant d'indice, alors que les appels récursifs les sélectionne par ordre décroissant.

• Au final cela ne change pas grand chose : on a quand même parcouru le graphe en profondeur...

## GRAPHES : RECHERCHE DE CYCLES (graphes connexes)

### ▶ Dans un graphe orienté

• L'algorithme ci-dessous détecte la présence d'un chemin entre deux sommets $S_1$ et $S_2$.
Au cours d'un parcours de graphe (en profondeur itératif avec pile ici) en partant de $S_1$, on teste si l'un des voisins rencontré est $S_2$, si c'est le cas on renvoie `True`, sinon `False` à la fin.

```
def ExisteChemin(G,S1,S2):
  Vu,Parcours,pile={},[],[S1]
  while len(pile)!=0:
    s=pile.pop()
    if s not in Vu:
      Vu[s]=True
      Parcours.append(s)
      for v in Voisins(G,s):
        if v==S2:
          return True
        elif v not in Vu:
          pile.append(v)
  return False
```

Pour détecter si un graphe orienté contient un cycle :

```
def CycliqueOriente(G):
  for S in G: # Si G dico d'adjacence
    if ExisteChemin(G,S,S):
    return True
  return False
```

Cela fonctionne car dans un graphe orienté chaque arc ne peut être parcouru que dans un sens et donc le même arc ne sera parcouru qu'une seule fois.

### ▶ Dans un graphe non orienté

Attention : lors d'un parcours de graphe une arête peut être parcourue dans un sens puis dans l'autre (pas autorisé pour un chemin simple et donc un cycle).
L'algorithme sur les graphes orientés ne fonctionne plus.
On ajoute à l'algorithme de parcours un dictionnaire des prédécesseurs de chaque sommet visité (`clef` = sommet visité, `valeur` = sommet qui a permis de découvrir ce sommet par la méthode de parcours choisie).

```
def ExisteCycle(G,S):
  Vu = {}
  pile=[S]
  pred={S:S}
  while len(pile)!=0:
    s = pile.pop()
    if s not in Vu:
      Vu[s]=True
      for v in Voisins(G,s):
        if v not in Vu:
          pile.append(v)
          pred[v]=s
        if S in Voisins(G,v):
          if pred[v]!=S:
            return True
    return False
```

**Idée :** On parcourt le graphe et si, lorsque qu'on est à un sommet `s` quelconque, on retombe sur un sommet voisin `v` déjà visité et que celui-ci n'a pas permis de découvrir `s` (arête pas déjà parcourue dans l'autre sens !) alors on a trouvé un cycle de `G` contenant `v` (et `s`).

# GRAPHES : PLUS COURT CHEMIN

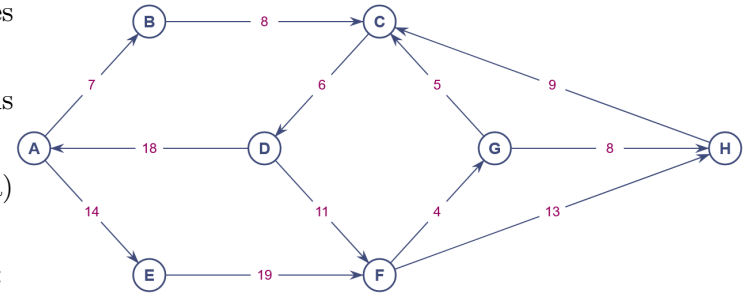
## ALGORITHME DE DIJKSTRA

► Un **graphe pondéré** $G$ est un graphe dont les arêtes ont été affectées d'un nombre appelé **poids**.

► La **longueur** d'un chemin devient la somme des poids de chacun des arcs/arêtes qui le constituent.

► La distance entre 2 sommets est la longueur du (de la) plus court(e) chemin (chaine) les reliant.

**Exemple :** La longueur du chemin $A-E-F-G$ est : $\ell = 14+19+4 = 37$.




**Graphe pondéré**

► **Principe de l'algorithme de Dijkstra** (graphes simples et poids des arcs positifs).

- Initialisation : À chaque sommet on attribue un poids $\infty$ (pas atteints) et 0 pour le sommet de départ.
- Si le plus court chemin reliant le sommet de départ $s_0$ à un autre sommet $S$ passe par les sommets $s_1$, $s_2$, ..., $s_k$ alors, les différentes étapes sont aussi les plus courts chemins reliant $s_0$ à ces sommets.
- À chaque étape, on choisit un sommet $s_k$ du graphe parmi ceux qui n'ont pas encore été atteints tel que la longueur connue provisoirement du plus court chemin allant de $s_0$ à $s_k$ soit la plus courte possible.

**Algorithme de Dijkstra partant du sommet $A$ en notant les prédécesseurs.**

| $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ | $H$ | Sommets sélectionnés |
|---|---|---|---|---|---|---|---|---|
| 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $A$ (0) |
| | 7 ($A$) | $\infty$ | $\infty$ | 14 ($A$) | $\infty$ | $\infty$ | $\infty$ | $B$ (7) |
| | | 15 ($B$) | $\infty$ | 14 ($A$) | $\infty$ | $\infty$ | $\infty$ | $E$ (14) |
| | | 15 ($B$) | $\infty$ | | 33 ($E$) | $\infty$ | $\infty$ | $C$ (15) |
| | | | 21 ($C$) | | 33 ($E$) | $\infty$ | $\infty$ | $D$ (21) |
| | | | | | 32 ($D$) | $\infty$ | $\infty$ | $F$ (32) |
| | | | | | | 36 ($F$) | 45 ($F$) | $G$ (36) |
| | | | | | | | 44 ($G$) | $H$ (44) |

► **Lecture inverse du plus court chemin** entre $A$ et $H$ : $H-G-F-D-C-B-A$

**Initialisation :**
• Dictionnaire `D` avec poids $\infty$ pour les sommets et un poids à 0 pour le sommet de départ `dep`.
• la liste des sommets traités `Vu=[]`.

**Itération :** Tant qu'il reste des sommets à voir,
• Sélection d'un nouveau sommet `S` de poids minimum non vu et ajout de `S` à `Vu`
• Création de la liste des voisins $v$ de `S`,
• pour chaque voisin $v$ pas encore traité,
si $d(dep,v) > d(dep,S)+d(S,v)$ alors `D[v]` est actualisée et `S` est le prédécesseur de `v`.

• **Fin :** Renvoi du dictionnaire `D` contenant donc les distances du sommet initial à chaque sommet et du dictionnaire `P` des prédécesseurs.

```
def Dijkstra(G:[[int]],dep:int)->(dict,dict):
    n = len(G)
    D = {k:np.inf for k in range(n)}
    D[dep] = 0
    Vu,P = {},{}
    while len(Vu)<n:
        m = np.inf
        for k in D:
            if D[k]<m and k not in Vu:
                m = D[k]
                S = k
        Vu[S] = True
        for v in Voisins(G,S):
            if v not in Vu and D[v]>D[S]+G[S][v]:
                D[v] = D[S]+G[S][v]
                P[v] = S
    return D,P
```

## ALGORITHME A$^\star$

▶ L' **algorithme** $A^\star$ a été développé en 1968 (Dijkstra en 1959) par Peter Hart, Nils Nilsson et Bertram Raphael.

▶ $A^\star$ calcule le plus court chemin entre une source et une UNIQUE destination (contrairement à Dijkstra).

▶ $A^\star$ dépend d'une certaine **heuristique** adaptée au problème à résoudre, basée sur des règles **empiriques** (qui reposent sur l'expérience) permettant d'accélérer la résolution du problème de façon simple (et sera meilleur que Dijkstra dans la majeure partie des cas).

▶ En général on utilise $A^\star$ sur des situations ou les sommets peuvent être placé sur une « carte » ce qui permet de définir une **distance à vol d'oiseau** entre deux sommets calculée par une fonction `Oiseau`.

▶ Cette **distance à vol d'oiseau** permet d'estimer la distance qu'il reste à parcourir pour rejoindre le sommet d'arrivée (en gros vérifier qu'on se dirige dans la « bonne direction »).

▶ On l'obtient en modifiant légèrement l'algorithme de Dijkstra :
- Déjà, on s'arrête dès qu'on a atteint le sommet « destination » souhaité !
- A chaque étape, parmi les sommets `v` pas encore vus et voisins du sommet `s` sélectionné, on retient celui qui minimise la quantité :

$$\boxed{h(\texttt{v}) = d(\texttt{dep},\texttt{v}) + \texttt{Oiseau}(\texttt{v},\texttt{arr})}$$

où $d(\texttt{dep},\texttt{v})$ est la distance entre le sommet de départ `dep` et le sommet `v` (*donnée présente dans le dictionnaire des distances de l'algorithme de Dijkstra*).

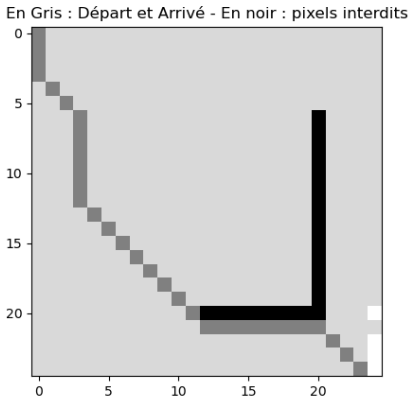
▶ La quantité $h(\texttt{s})$ est donc l'**heuristique** que l'on cherche à minimiser.

▶ Dans la pratique, $A^\star$ est bien plus rapide que l'algorithme de Dijkstra, notamment sur des graphes d'ordre très grand car on visite beaucoup moins de sommets.

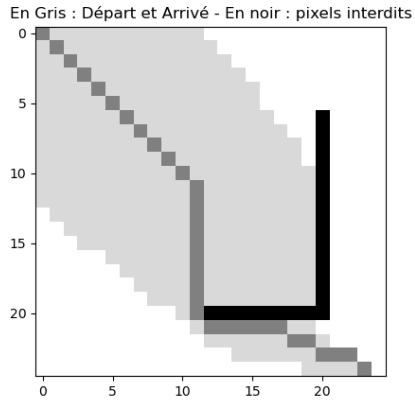
**Exemple.** Voici un exemple comparatif entre l'algorithme de Dijkstra et $A^\star$. On cherche à rallier le sommet de départ (en haut à gauche) au sommet d'arrivée (en bas à droite) en :

— ne passant pas sur les cases noires (mur), considérant qu'une case (pas au bord de la grille) a 8 voisins,

— donc en s'autorisant les déplacements horizontaux et verticaux entre deux cases (de longueur 1) mais aussi en diagonale (longueur $\sqrt{2}$)




**Parcours dans la grille avec Dijkstra**




**Parcours dans la grille avec $A^\star$**

En gris clair les pixels visités par les deux algorithmes, en gris foncé un chemin le plus court entre le départ et l'arrivée. On voit bien que l'algorithme $A^\star$ visite moins de sommets et dès qu'il a trouvé un moyen de contourner le `mur de pixels`, file dans la direction de l'arrivée sans tester beaucoup de pixels.

▶ Voici un exemple de programmation Python de l'algorithme $A^\star$ adapté de Dijkstra. On n'explique que les différences par rapport à l'algorithme de Dijkstra.

```
def A_Star(G:[[int]],dep:int,arr:int): # G sous forme de matrice d'adjacence
  import numpy as np
  Dist = {dep:0}
  H={} # Dictionnaire des heuristiques : H[s]=Dist[s]+Oiseau[s]
  H[dep] = Oiseau(dep,arr) # Initialisation du dictionnaire
  Pred = {}
  Vu = []
  while S != arr:
    S = MiniDistNonVu(H,Vu) # Sélection du sommet d'heuristique minimale
    Vu.append(S)
    for v in Voisins(G,S):
      if v not in Vu:
        if v not D or Dist[v] > Dist[S] + G[S][v]:
          Dist[v] = Dist[S] + G[S][v]
          Pred[v] = S # Mise à jour des prédécesseurs
          H[v] = Dist[v] + Oiseau(v,arr) # Mise à jour des heuristiques
  return Dist,Pred
```

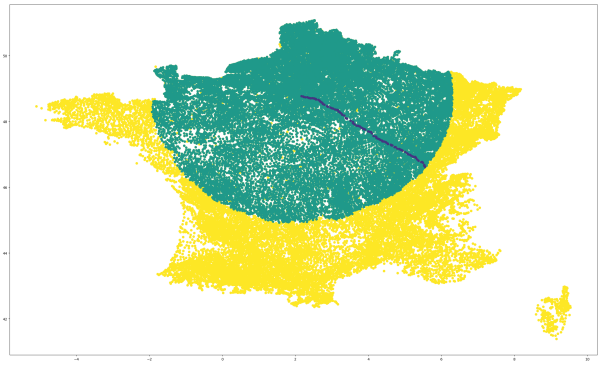
▶ Cet algorithme fait appel à trois fonctions : `Voisins`, `MiniDistNonVu` et `Oiseau` dont la syntaxe dépend grandement du contexte du problème.

— `Voisins(G,S)` renvoie la liste des sommets voisins de `s` dans `G`,

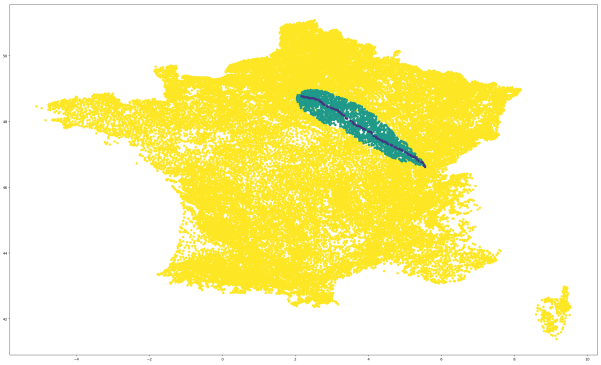
— `MiniDistNonVu(H,Vu)` renvoie le sommet non visité (`not in Vu` dont l'heuristique dans `H` est minimale (donc la clé de valeur minimale dans `H`) qui n'est pas dans `Vu`).

— `Oiseau(v,arr)` calcule la distance « à vol d'oiseau » restant à parcourir depuis `v` jusqu'à l'arrivée `arr`.

**Exemple.**
Un dernier exemple pour comparer les algorithmes de Dijkstra et $A^\star$. Il s'agit de minimiser (en km) un trajet routier entre deux communes de France. On dispose des coordonnées GPS de chaque communes et de la liste des communes qui sont voisines les unes des autres (donc reliées par une route). On chercher un trajet entre les deux communes en ne passant que par des communes voisines.



**Parcours sur la carte de France avec Dijkstra**



**Parcours sur la carte de France avec $A^\star$**

En jaune (gris clair) : les communes de France, en turquoise (gris moyen) les communes visitées et bleu (gris foncé) un chemin le plus court trouvé.

On s'aperçoit que l'algorithme de Dijkstra cherche le chemin le plus court sans se soucier de la « direction » à privilégier pour rallier la commune d'arrivée alors qu'avec l'algorithme $A^\star$ il y a très peu de communes testées.